



Real-Time Operating Systems for  
ARM Cortex-M Microcontrollers

# EMBEDDED SYSTEMS

JONATHAN VALVANO

# EMBEDDED SYSTEMS:

---

## REAL-TIME OPERATING SYSTEMS FOR ARM CORTEX-M MICROCONTROLLERS

Volume 3

*Fourth Edition,*

*January 2017*

Jonathan W. Valvano

# *Fourth edition*

## *January 2017*

ARM and uVision are registered trademarks of ARM Limited.

Cortex and Keil are trademarks of ARM Limited.

Stellaris and Tiva are registered trademarks Texas Instruments.

Code Composer Studio is a trademark of Texas Instruments.

All other product or service names mentioned herein are the trademarks of their respective owners.

In order to reduce costs, this college textbook has been self-published. For more information about my classes, my research, and my books, see <http://users.ece.utexas.edu/~valvano/>

For corrections and comments, please contact me at: [valvano@mail.utexas.edu](mailto:valvano@mail.utexas.edu). Please cite this book as: J. W. Valvano, Embedded Systems: Real-Time Operating Systems for ARM® Cortex™-M Microcontrollers, Volume 3, <http://users.ece.utexas.edu/~valvano/>, ISBN: 978-1466468863.

Copyright © 2017 Jonathan W. Valvano

All rights reserved. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, web distribution, information networks, or information storage and retrieval, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

ISBN-13: 978-1466468863

ISBN-10: 1466468866

# Table of Contents

[Preface to The Fourth Edition](#)

[Preface to Volume 3](#)

[Acknowledgements](#)

[1. Computer Architecture](#)

[1.1. Introduction to Real-Time Operating Systems](#)

[1.1.1. Real-time operating systems](#)

[1.1.2. Embedded Systems](#)

[1.2. Computer Architecture](#)

[1.2.1. Computers, processors, and microcontrollers](#)

[1.2.2. Memory](#)

[1.3. Cortex-M Processor Architecture](#)

[1.3.1. Registers](#)

[1.3.2. Stack](#)

[1.3.3. Operating modes](#)

[1.3.4. Reset](#)

[1.3.5. Clock system](#)

[1.4. Texas Instruments Cortex-M Microcontrollers](#)

[1.4.1. Introduction to I/O](#)

[1.4.2. Texas Instruments TM4C123 LaunchPad I/O pins](#)

[1.4.3. Texas Instruments TM4C1294 Connected LaunchPad I/O pins](#)

[1.4.4. Texas Instruments MSP432 LaunchPad I/O pins](#)

[1.4.5. Interfacing to a LaunchPad](#)

[1.5. ARM Cortex-M Assembly Language](#)

[1.5.1. Syntax](#)

[1.5.2. Addressing modes and operands](#)

[1.5.3. List of twelve instructions](#)

[1.5.4. Accessing memory](#)



### 1.5.5. Functions

### 1.5.6. ARM Cortex Microcontroller Software Interface Standard

### 1.5.7. Conditional execution

### 1.5.8. Stack usage

### 1.5.9. Floating-point math

### 1.5.10. Keil assembler directives

## 1.6. Pointers in C

### 1.6.1. Pointers

### 1.6.2. Arrays

### 1.6.3. Linked lists

## 1.7. Memory Management

### 1.7.1. Use of the heap

### 1.7.2. Simple fixed-size heap

### 1.7.3. Memory manager: malloc and free

## 1.8. Introduction to debugging

## 1.9. Exercises

# 2. Microcontroller Input/Output

## 2.1. Parallel I/O

### 2.1.1. TM4C I/O programming

### 2.1.2. MSP432 I/O programming

## 2.2. Interrupts

### 2.2.1. NVIC

### 2.2.2. SysTick periodic interrupts

### 2.2.3. Periodic timer interrupts

### 2.2.4. Critical sections

### 2.2.5. Executing periodic tasks

### 2.2.6. Software interrupts

## 2.3. First in First Out (FIFO) Queues

## 2.4. Edge-triggered Interrupts

### 2.4.1. Edge-triggered interrupts on the TM4C123

### 2.4.2. Edge-triggered Interrupts on the MSP432

## 2.5. UART Interface

### 2.5.1. Transmitting in asynchronous mode

### 2.5.2. Receiving in asynchronous mode

### 2.5.3. Interrupt-driven UART on the TM4C123

### 2.5.4. Interrupt-driven UART on the MSP432

## 2.6. Synchronous Transmission and Receiving using the SSI

## 2.7. Input Capture or Input Edge Time Mode

### 2.7.1. Basic principles

### 2.7.2. Period measurement on the TM4C123

### 2.7.3. Period measurement on the MSP432

### 2.7.4. Pulse width measurement

### 2.7.5. Ultrasonic distance measurement

## 2.8. Pulse Width Modulation

### 2.8.1. Pulse width modulation on the TM4C123

### 2.8.2. Pulse width modulation on the MSP432

## 2.9. Analog Output

## 2.10. Analog Input

### 2.10.1. ADC Parameters

### 2.10.2. Internal ADC on TM4C

### 2.10.3. Internal ADC on MSP432

### 2.10.4. IR distance measurement

## 2.11. OS Considerations for I/O Devices

### 2.11.1 Board Support Package

### 2.11.2 Path Expression

## 2.12. Debugging

### 2.12.1. Functional Debugging

### 2.12.2. Performance Debugging (FFT analysis)

### 2.12.3. Debugging heartbeat

### 2.12.4. Profiling

## 2.13. Exercises

## 3. Thread Management

## 3.1. Introduction to RTOS

### 3.1.1. Motivation

### 3.1.2. Parallel, distributed and concurrent programming

### 3.1.3. Introduction to threads

### 3.1.4. States of a main thread

### 3.1.5. Real-time systems

### 3.1.6. Producer/Consumer problem using a mailbox

### 3.1.7. Scheduler

## 3.2. Function pointers

## 3.3. Thread Management

### 3.3.1. Two types of threads

### 3.3.2. Thread Control Block (TCB)

### 3.3.3. Creation of threads

### 3.3.4. Launching the OS

### 3.3.5. Switching threads

### 3.3.6. Profiling the OS

### 3.3.7. Linking assembly to C

### 3.3.8. Periodic tasks

## 3.4. Semaphores

## 3.5. Thread Synchronization

### 3.5.1. Resource sharing, nonreentrant code or mutual exclusion

### 3.5.2. Condition variable

### 3.5.3. Thread communication between two threads using a mailbox

## 3.6. Process Management

## 3.7. Dynamic loading and linking

## 3.8. Exercises

## 4. Time Management

### 4.1. Cooperation

#### 4.1.1. Spin-lock semaphore implementation with cooperation

#### 4.1.2. Cooperative Scheduler

### 4.2. Blocking semaphores

**4.2.1. The need for blocking**

**4.2.2. The blocked state**

**4.2.3. Implementation**

**4.2.4. Thread rendezvous**

**4.3. First In First Out Queue**

**4.3.1. Producer/Consumer problem using a FIFO**

**4.3.2. Little's Theorem**

**4.3.3. FIFO implementation**

**4.3.4. Three-semaphore FIFO implementation**

**4.3.5. Two-semaphore FIFO implementation**

**4.3.6. One-semaphore FIFO implementation**

**4.3.7. Kahn Process Networks**

**4.4. Thread sleeping**

**4.5. Deadlocks**

**4.6. Monitors**

**4.7. Fixed Scheduling**

**4.8. Exercises**

**5. Real-time Systems**

**5.1. Data Acquisition Systems**

**5.1.1. Approach**

**5.1.2. Performance Metrics**

**5.1.3. Audio Input/Output**

**5.2. Priority scheduler**

**5.2.1. Implementation**

**5.2.2. Multi-level Feedback Queue**

**5.2.3. Starvation and aging**

**5.2.4. Priority inversion and inheritance on Mars Pathfinder**

**5.3. Debouncing a switch**

**5.3.1. Approach to debouncing**

**5.3.2. Debouncing a switch on TM4C123**

**5.3.3. Debouncing a switch on MSP432**

5.4. Running event threads as high priority main threads

5.5. Available RTOS

5.5.1. Micrium uC/OS-II

5.5.2. Texas Instruments RTOS

5.5.3. ARM RTX Real-Time Operating System

5.5.4. FreeRTOS

5.5.5. Other Real Time Operating Systems

5.6. Exercises

6. Digital Signal Processing

6.1. Basic Principles

6.2. Multiple Access Circular Queue

6.3. Using the Z-Transform to Derive Filter Response

6.4. IIR Filter Design Using the Pole-Zero Plot

6.5. Discrete Fourier Transform

6.6. FIR Filter Design

6.7. Direct-Form Implementations.

6.8. Exercises

7. High-Speed Interfacing

7.1. The Need for Speed

7.2. High-Speed I/O Applications

7.3. General Approaches to High-Speed Interfaces

7.3.1. Hardware FIFO

7.3.2. Dual Port Memory

7.3.3. Bank-Switched Memory

7.4. Fundamental Approach to DMA

7.4.1. DMA Cycles

7.4.2. DMA Initiation

7.4.3. Burst versus Single Cycle DMA

7.4.4. Single Address versus Dual Address DMA

7.4.5. DMA programming on the TM4C123

7.6. Exercises



## 8. File system management

### 8.1. Performance Metrics

#### **8.1.1. Usage**

#### **8.1.2. Specifications**

#### **8.1.3. Fragmentation**

### 8.2. File System Allocation

#### **8.2.1. Contiguous allocation**

#### **8.2.2. Linked allocation**

#### **8.2.3. Indexed allocation**

#### **8.2.4. File allocation table (FAT)**

### 8.3. Solid State Disk

#### **8.3.1. Flash memory**

#### **8.3.2. Flash device driver**

#### **8.3.3. eDisk device driver**

#### **8.3.4. Secure digital card interface**

### 8.4. Simple File System

#### **8.4.1. Directory**

#### **8.4.2. Allocation**

#### **8.4.3. Free space management**

### 8.5. Write-once File System

#### **8.5.1. Usage**

#### **8.5.2. Allocation**

#### **8.5.3. Directory**

#### **8.5.4. Append**

#### **8.5.5. Free space management**

### 8.6. Readers-Writers Problem

### 8.7. Exercises

## 9. Communication Systems

### 9.1. Fundamentals

#### **9.1.1. The network**

#### **9.1.2. Physical Channel**

### **9.1.3. Wireless Communication**

### **9.1.4. Radio**

## **9.2. Controller Area Network (CAN)**

### **9.2.1. The Fundamentals of CAN**

### **9.2.2. Texas Instruments TM4C CAN**

## **9.3. Embedded Internet**

### **9.3.1. Abstraction**

### **9.3.2. Message Protocols**

### **9.3.3. Ethernet Physical Layer**

### **9.3.4. Ethernet on the TM4C1294**

## **9.4. Internet of Things**

### **9.4.1. Basic Concepts**

### **9.4.2. UDP and TCP Packets**

### **9.4.3. Web server**

### **9.4.4. UDP communication over WiFi**

### **9.4.5. Other CC3100 Applications**

## **9.4. Bluetooth Fundamentals**

### **9.4.1. Bluetooth Protocol Stack**

### **9.4.2. Client-server Paradigm**

## **9.5. CC2650 Solutions**

### **9.5.1. CC2650 Microcontroller**

### **9.5.2. Single Chip Solution, CC2650 LaunchPad**

## **9.6. Network Processor Interface (NPI)**

### **9.6.1. Overview**

### **9.6.2. Services and Characteristics**

### **9.6.3. Advertising**

### **9.6.4. Read and Write Indications**

## **9.7. Application Layer Protocols for Embedded Systems**

### **9.7.1. CoAP**

### **9.7.2 MQTT**

## **9.8. Exercises**

## 10. Robotic Systems

### 10.1. Introduction to Digital Control Systems

### 10.2. Binary Actuators

#### **10.2.1. Electrical Interface**

#### **10.2.2. DC Motor Interface with PWM**

### 10.3. Sensors

### 10.4. Odometry

### 10.5. Simple Closed-Loop Control Systems.

### 10.6. PID Controllers

#### **10.6.1. General Approach to a PID Controller**

#### **10.6.2. Design Process for a PID Controller**

### 10.7. Fuzzy Logic Control

### 10.8. Exercises

## Appendix 1. Glossary

## Appendix 2. Solutions to Checkpoints

## Reference Material

## **Preface to The Fourth Edition**

There are two major additions to this fourth edition. First, this version supports both the TM4C and the MSP432 architectures. The material for the LM3S series has been removed. Volumes 1 and 2 focused on the hardware and software aspects I/O interfacing. In this volume we provide a set of low level device drivers allowing this volume to focus on real-time operating systems, digital signal processing, control systems, and the internet of things. The second addition is Bluetooth Low Energy (BLE), which will be implemented by interfacing a CC2650, in a similar manner with which IEEE802.11b wifi is implemented in this book using the CC3100. Running on the CC2650 will be an application programmer interface called Simple Network Processor (SNP). SNP allows the TM4C123/MSP432 microcontroller to implement BLE using a simple set of UART messaging. Off-loading the BLE functions to the CC2650 allows the target microcontroller to implement system level functions without the burden of satisfying the real-time communication required by Bluetooth.

## Preface to Volume 3

Embedded systems are a ubiquitous component of our everyday lives. We interact with hundreds of tiny computers every day that are embedded into our houses, our cars, our toys, and our work. As our world has become more complex, so have the capabilities of the microcontrollers embedded into our devices. The ARM Cortex-M family represents the new class of microcontrollers much more powerful than the devices available ten years ago. The purpose of this book is to present the design methodology to train young engineers to understand the basic building blocks that comprise devices like a cell phone, an MP3 player, a pacemaker, antilock brakes, and an engine controller.

This book is the third in a series of three books that teach the fundamentals of embedded systems as applied to the ARM Cortex-M family of microcontrollers. This third volume is primarily written for senior undergraduate or first-year graduate electrical and computer engineering students. It could also be used for professionals wishing to design or deploy a real-time operating system onto an ARM platform. The first book Embedded Systems: Introduction to ARM Cortex-M Microcontrollers is an introduction to computers and interfacing focusing on assembly language and C programming. The second book Embedded Systems: Real-Time Interfacing to ARM Cortex-M Microcontrollers focuses on interfacing and the design of embedded systems. This third book is an advanced book focusing on operating systems, high-speed interfacing, control systems, and robotics.

An embedded system is a system that performs a specific task and has a computer embedded inside. A system is comprised of components and interfaces connected together for a common purpose. This book presents components, interfaces and methodologies for building systems. Specific topics include microcontrollers, design, verification, hardware/software synchronization, interfacing devices to the computer, timing diagrams, real-time operating systems, data collection and processing, motor control, analog filters, digital filters, and real-time signal processing.

In general, the area of embedded systems is an important and growing discipline within electrical and computer engineering. In the past, the educational market of embedded systems has been dominated by simple microcontrollers like the PIC, the 9S12, and the 8051. This is because of their market share, low cost, and historical dominance. However, as problems become more complex, so must the systems that solve them. A number of embedded system paradigms must shift in order to accommodate this growth in complexity. First, the number of calculations per second will increase from millions/sec to billions/sec. Similarly, the number of lines of software code will also increase from thousands to millions. Thirdly, systems will involve multiple microcontrollers supporting many simultaneous operations. Lastly, the need for system verification will continue to grow as these systems are deployed into safety critical applications. These changes are more than a simple growth in size and bandwidth. These systems must employ parallel programming, high-speed synchronization, real-time operating systems, fault tolerant design, priority interrupt



handling, and networking. Consequently, it will be important to provide our students with these types of design experiences. The ARM platform is both low cost and provides the high performance features required in future embedded systems. Although the ARM market share is large and will continue to grow. Furthermore, students trained on the ARM will be equipped to design systems across the complete spectrum from simple to complex. The purpose of writing these three books at this time is to bring engineering education into the 21<sup>st</sup> century.

This book employs many approaches to learning. It will not include an exhaustive recapitulation of the information in data sheets. First, it begins with basic fundamentals, which allows the reader to solve new problems with new technology. Second, the book presents many detailed design examples. These examples illustrate the process of design. There are multiple structural components that assist learning. Checkpoints, with answers in the back, are short easy to answer questions providing immediate feedback while reading. Homework problems, which typically are simpler than labs, provide more learning opportunities. The book includes an index and a glossary so that information can be searched. The most important learning experiences in a class like this are of course the laboratories. More detailed lab descriptions are available on the web. Specifically for Volume 1, look at the lab assignments for EE319K. For Volume 2 refer to the EE445L labs, and for this volume, look at the lab assignments for EE445M/EE380L.6.

There is a web site accompanying this book <http://users.ece.utexas.edu/~valvano/arm>. Posted here are ARM Keil™ uVision® and Texas Instruments Code Composer Studio™ projects for each of the example programs in the book. You will also find data sheets and Excel spreadsheets relevant to the material in this book.

The book will cover embedded systems for ARM® Cortex™-M microcontrollers with specific details on the TM4C123, TM4C1294, and MSP432. Most of the topics can be run on any Texas Instruments Cortex M microcontroller. In these books the terms **MSP432** and **TM4C** will refer to any of the Texas Instruments ARM Cortex-M based microcontrollers. Although the solutions are specific for the **MSP432** and **TM4C** families, it will be possible to use these books for other ARM derivatives.

# Acknowledgements

I owe a wonderful debt of gratitude to Daniel Valvano. He wrote and tested most of the software examples found in these books. Secondly, he maintains the example web site, <http://users.ece.utexas.edu/~valvano/arm>. Lastly, he meticulously proofread this manuscript.

Many shared experiences contributed to the development of this book. First I would like to acknowledge the many excellent teaching assistants I have had the pleasure of working with. Some of these hard-working, underpaid warriors include Pankaj Bishnoi, Rajeev Sethia, Adson da Rocha, Bao Hua, Raj Randeri, Santosh Jodh, Naresh Bhavaraju, Ashutosh Kulkarni, Bryan Stiles, V. Krishnamurthy, Paul Johnson, Craig Kochis, Sean Askew, George Panayi, Jeehyun Kim, Vikram Godbole, Andres Zambrano, Ann Meyer, Hyunjin Shin, Anand Rajan, Anil Kottam, Chia-ling Wei, Jignesh Shah, Icaro Santos, David Altman, Nachiket Kharalkar, Robin Tsang, Byung Geun Jun, John Porterfield, Daniel Fernandez, Deepak Panwar, Jacob Egner, Sandy Hermawan, Usman Tariq, Sterling Wei, Seil Oh, Antonius Keddis, Lev Shuhatovich, Glen Rhodes, Geoffrey Luke, Karthik Sankar, Tim Van Ruitenbeek, Raffaele Cetrulo, Harshad Desai, Justin Capogna, Arindam Goswami, Jungho Jo, Mehmet Basoglu, Kathryn Loeffler, Evgeni Krimer, Nachiappan Valliappan, Razik Ahmed, Sundeep Korrapati, Song Zhang, Zahidul Haq, Matthew Halpern, Cruz Monrreal II, Pohan Wu, Saugata Bhattacharyya, Dayo Lawal, Abhishek Agarwal, Sparsh Singhai, Nagaraja Revanna, Mahesh Srinivasan, Victoria Bill, Alex Hsu, Dylan Zika, Chun-Kai Chang, Zhao Zheng, Ce Wei, Kelsey Taylor Ball, Brandon Nguyen, Turan Vural, Schuyler Christensen, Danny Vo, Justin Nguyen, Danial Rizvi, Armand Behroozi, Vivian Tan, Anthony Bauer, Jun Qi Lau, Corey Cormier, Cody Horton, Youngchun Kim, Ryan Chow, Cody Horton, Corey Cormier, and Dylan Zika. These teaching assistants have contributed greatly to the contents of this book and particularly to its laboratory assignments. Since 1981, I estimate I have taught embedded systems to over 5000 students. My students have recharged my energy each semester with their enthusiasm, dedication, and quest for knowledge. I have decided not to acknowledge them all individually. However, they know I feel privileged to have had this opportunity.

Next, I appreciate the patience and expertise of my fellow faculty members here at the University of Texas at Austin. From a personal perspective Dr. John Pearce provided much needed encouragement and support throughout my career. Over the last few years, I have enjoyed teaching embedded systems with Drs. Ramesh Yerraballi, Mattan Erez, Andreas Gerstlauer, and William Bard. Bill has contributed to both the excitement and substance of our laboratory based on this book. Many of the suggestions and corrections from Chris Shore and Drew Barbier of ARM about Volume 1 applied equally to this volume. Austin Blackstone created and debugged the Code Composer Studio™ versions of the example programs posted on the web. Austin also taught me how to run the CC3000 and CC3100 Wifi examples on the LaunchPad.

Ramesh Yerraballi and I have created two MOOCs, which have had over 110,000 students, and delivered to 110 countries. The new material in this book was developed under the watchful eye of Professor Yerraballi. It has been an honor and privilege to work with such a skilled and dedicated educator.

Andreas Gerstlauer has taught a course based on this book multiple times, and I have incorporated many of his ideas into this edition of the book. Furthermore, you will find a rich set of material if you search with these keywords **Gerstlauer RTOS utexas**.

Sincerely, I appreciate the valuable lessons of character and commitment taught to me by my parents and grandparents. I recall how hard my parents and grandparents worked to make the world a better place for the next generation. Most significantly, I acknowledge the love, patience and support of my wife, Barbara, and my children, Ben Dan and Liz. In particular, Dan designed and tested most of the MSP432 and TM4C software presented in this book.

By the grace of God, I am truly the happiest man on the planet, because I am surrounded by these fine people.

*Jonathan W. Valvano*

Good luck

# 1. Computer Architecture

## Chapter 1 objectives are to:

- Present a brief review of computer architecture
- Overview the ARM® Cortex™ -M processor including assembly language
- Introduce the Texas Instruments MSP432/TM4C family of microcontrollers

The overall objective of this book is to teach the design of real-time operating systems for embedded systems. We define a system as real time if there is a small and bounded delay between the time when a task should be completed and when it is actually completed. We will present both fundamental principles and practical solutions. Interfacing to the microcontroller was presented in detail in Volume 2 and reviewed in the first two chapters of this book. The overlap allows this book to stand alone as a text to teach embedded real time operating systems. This first chapter will review the architecture of the Texas Instruments MSP432/TM4C family of microcontrollers. When designing operating systems, we need to understand the details of the architecture. In particular, we must perform many functions in assembly language. Furthermore, managing memory will require an intimate understanding of how the processor accesses memory at the most basic level.

# 1.1. Introduction to Real-Time Operating Systems

## 1.1.1. Real-time operating systems

A computer system has many types of resources such as memory, I/O, data, and processors. A **real-time operating system** (RTOS) is software that manages these resources, guaranteeing all timing constraints are satisfied. Figure 1.1 illustrates the relationship between hardware and software. On the left is a basic system without an operating system. Software is written by a single vendor for a specific microcontroller. As the system becomes more complex (middle figure), an operating system facilitates the integration of software from multiple vendors. By providing a **hardware abstraction layer** (HAL) an operating system simplifies porting application code from one microcontroller to another. In order to provide additional processing power, embedded systems of the future will require multiple microcontrollers, processors with specialized coprocessors and/or a microcontroller with multiple cores (right figure). Synchronization and assigning tasks across distributed processors are important factors. As these systems become more complex, the role of the operating system will be increasingly important.

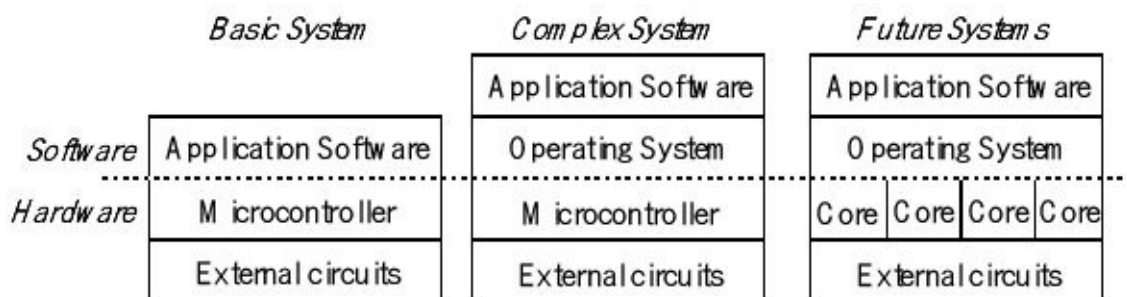


Figure 1.1. An operating system is a software layer between the application software and the hardware.

The RTOS must **manage resources** like memory, processor and I/O. The RTOS will **guarantee strict timing constraints** and provide **reliable** operation. The RTOS will support **synchronization** and **communication** between tasks. As complex systems are built the RTOS manages the **integration of components**. **Evolution** is the notion of a system changing to improve performance, features and reliability. The RTOS must manage change. When designing a new system, it is good design practice to build a new system by changing an existing system. The notion of **portability** is the ease at which one system can be changed or adapted to create another system.

The **response time** or **latency** is the delay from a request to the beginning of the service of that request. There are many definitions of bandwidth. In this book we define **bandwidth** as the number of information bytes/sec that can be transferred or



processed. We can compare and contrast regular operating systems with real-time operating systems.

<i>Regular OS</i>	<i>Real-time OS</i>
Complex	Simple
Best effort	Guaranteed response
Fairness	Strict timing constraints
Average bandwidth	Minimum and maximum limits
Unknown components	Known components
Unpredictable behavior	Predictable behavior
Plug and play	Upgradable

**Table 1.1. Comparison of regular and real-time operating systems.**

From Table 1.1 we see that real-time operating systems have to be simple so they may be predictable. While traditional operating systems gauge their performance in terms of response time and fairness, real-time operating systems target strict timing constraints and upper, lower bounds on bandwidth. One can expect to know all the components of the system at design time and component changes happen much more infrequently.

**Checkpoint 1.1:** What does real time mean?

## 1.1.2. Embedded Systems

An **embedded system** is a smart device with a processor that has a special and dedicated purpose. The user usually does not or cannot upgrade the hardware/software or change what the system does. **Real time** means that the embedded system must respond to critical events within a strictly defined time, called the deadline. A guarantee to meet all deadlines can only be made if the behavior of the operating system can be predicted. In other words the timing must be deterministic. There are five types of software functions the processor can perform in an embedded system. Similar to a general-purpose computer, it can perform mathematical and/or data processing operations. It can analyze data and make decisions based on the data. A second type involves handling and managing time: as an input (e.g., measure period), an output (e.g., output waveforms), and a means to synchronize tasks (e.g., run 1000 times a second). A third type involves real-time input/output for the purpose of measurement or control. The fourth type involves digital signal processing (DSP), which are mathematical calculations on data streams. Examples include audio, video, radar, and sonar. The last type is communication and networking. As embedded systems become more complex, how the components are linked together will become increasingly important.

There are two classifications of embedded systems as shown in Figure 1.2. A **transformative system** collects data from inputs, makes decisions, and affects its environment by driving actuators. The robot systems presented in Chapter 10 are

examples of transformative systems. A **reactive system** collects data in a continuous fashion and produce outputs also in a continuous fashion. Digital signal processing algorithms presented in Chapter 6 are examples of reactive systems.

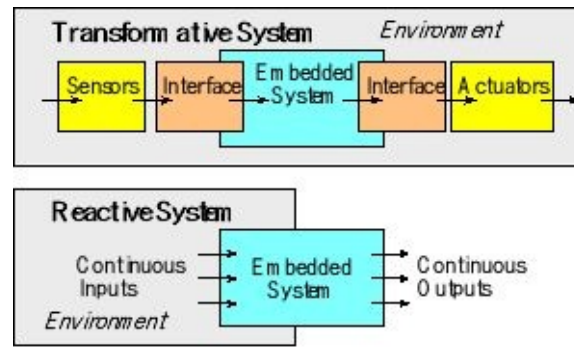


Figure 1.2. Embedded systems can transform or react to the environment.

Six **constraints** typify an embedded system. First, they are **small size**. For example, many systems must be handheld. Second, they must have **low weight**. If the device is deployed in a system that moves, e.g., attached to a human, aircraft or vehicle, then weight incurs an energy cost. Third, they often must be **low power**. For example, they might need to operate for a long time on battery power. Low power also impacts the amount of heat they are allowed to generate. Fourth, embedded systems often must operate in **harsh environments**, such as heat, pressure, vibrations, and shock. They may be subject to noisy power, RF interference, water, and chemicals. Fifth, embedded systems are often used in **safety critical systems**. Real-time behavior is essential. For these systems they must function properly at extremely high levels of reliability. Lastly, embedded systems are extremely **sensitive to cost**. Most applications are profit-driven. For high-volume systems a difference in pennies can significantly affect profit.

**Checkpoint 1.2:** What is an embedded system?

**Checkpoint 1.3:** List the six constraints typically found in an embedded system?

## 1.2. Computer Architecture

### 1.2.1. Computers, processors, and microcontrollers

Given that an operating system is a manager of resources provided by the underlying architecture, it would serve the reader well to get acquainted with the architecture the OS must manage. In this section we will delve into these details of the building blocks of computer architecture, followed by the specifics of the ARM Cortex M4 processor architecture, in particular TI's implementation of the ARM ISA found on the TM4C and MSP432.

A **computer** combines a central processing unit (CPU), random access memory (RAM), read only memory (ROM), and input/output (I/O) ports. The common bus in Figure 1.3 defines the von Neumann architecture. **Software** is an ordered sequence of very specific instructions that are stored in memory, defining exactly what and when certain tasks are to be performed.

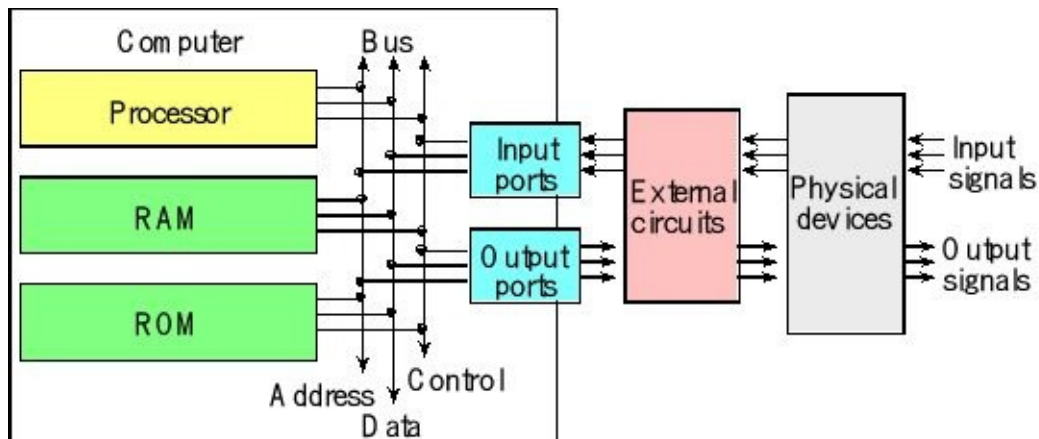


Figure 1.3. The basic components of a computer system include processor, memory and I/O.

The CPU or **processor** executes the software by retrieving (from memory) and interpreting these instructions one at a time. An ARM Cortex-M microcontroller includes a processor, memory and input/output. The processor, memory and peripherals are connected via multiple buses. Because instructions are fetched via the ICode bus and data are fetched via the System bus, the Cortex M is classified as a Harvard architecture. Having multiple busses allows the system to do several things simultaneously. For example, the processor could be reading an instruction from ROM using the ICode bus and writing data to RAM using the System bus.

The ARM Cortex-M processor has four major components, as illustrated in Figure 1.4. There are **bus interface units** (BIU) that read data from the bus during a read

cycle and write data onto the bus during a write cycle. The BIU always drives the address bus and the control signals of the bus. The **effective address register (EAR)** contains the memory address used to fetch the data needed for the current instruction. Cortex-M microcontrollers execute Thumb instructions extended with Thumb-2 technology. An overview of these instructions will be presented in Section 1.5. Many functions in an operating system will require detailed understanding of the architecture and assembly language.

The **control unit (CU)** orchestrates the sequence of operations in the processor. The CU issues commands to the other three components. The **instruction register (IR)** contains the operation code (or op code) for the current instruction. When extended with Thumb-2 technology, op codes are either 16 or 32 bits wide.

The **arithmetic logic unit (ALU)** performs arithmetic and logic operations. Addition, subtraction, multiplication and division are examples of arithmetic operations. Examples of logic operations are, and, or, exclusive-or, and shift. Many processors used in embedded applications support specialized operations such as table lookup, multiply and accumulate, and overflow detection.

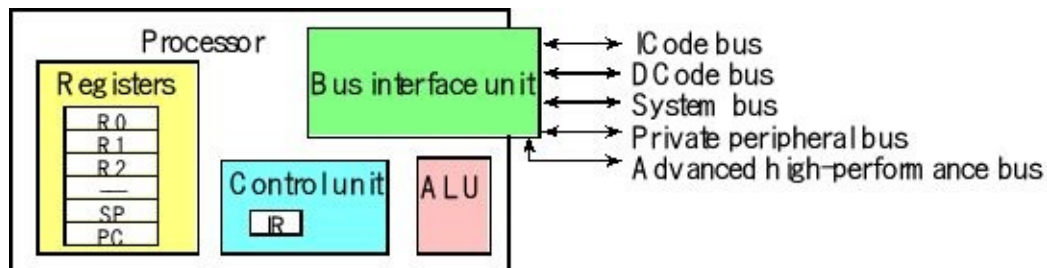


Figure 1.4. The four basic components of a processor.

A very small microcomputer, called a **microcontroller**, contains all the components of a computer (processor, memory, I/O) on a single chip. The Atmel ATtiny and the TI TM4C123 are examples of microcontrollers. Because a microcomputer is a small computer, this term can be confusing because it is used to describe a wide range of systems from a 6-pin ATtiny4 running at 1 MHz with 512 bytes of program memory to a personal computer with state-of-the-art 64-bit multi-core processor running at multi-GHz speeds having terabytes of storage.

An **application-specific integrated circuit (ASIC)** is digital logic that solves a very specific problem. See Figure 1.5. A **field-programmable gate array (FPGA)** is one approach to ASIC prototyping, allowing you to program and reprogram the digital logic. Verilog and VHDL are example FPGA programming environments. ASIC design is appropriate for problems defined with logic and/or numerical equations. On the other hand, microcontrollers are appropriate for problems solved with algorithms or sequential processes. Mature problems with high volume can create ASIC solutions directly as digital logic integrated circuits. On the other hand, microcontrollers can be used for low-volume problems and have the advantage of having a shorter time to market. Microcontrollers, because they are programmed with software, allow a flexibility to upgrade features, provide user-tailored performance,

and solve problems with uncertain or changing requirements. Some systems have both microcontrollers and ASICs.

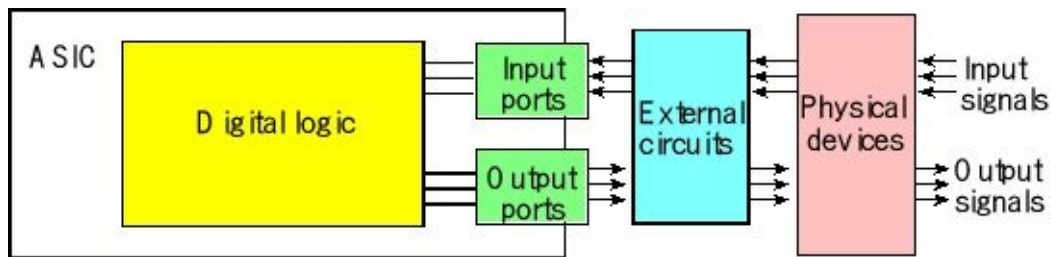


Figure 1.5. A system implemented with an ASIC and I/O.

In an embedded system the software is converted to machine code, which is a list of instructions, and stored in nonvolatile flash ROM. As instructions are fetched, they are placed in a **pipeline**. This allows instruction fetching to run ahead of execution. Instructions on the Cortex-M processor are fetched in order and executed in order. However, it can execute one instruction while fetching the next. Many high-speed processors allow out of order execution, support parallel execution on multiple cores, and employ branch prediction.

On the ARM Cortex-M processor, an instruction may read memory or write memory, but does not read and write memory in the same instruction. Each of the phases may require one or more bus cycles to complete. Each bus cycle reads or writes one piece of data. Because of the multiple bus architecture, most instructions execute in one or two cycles. For more information on the time to execute instructions, see Table 3.1 in the Cortex-M Technical Reference Manual.

Figure 1.6 shows a simplified block diagram of a microcontroller based on the ARM Cortex-M processor. It is a **Harvard architecture** because it has separate data and instruction buses.

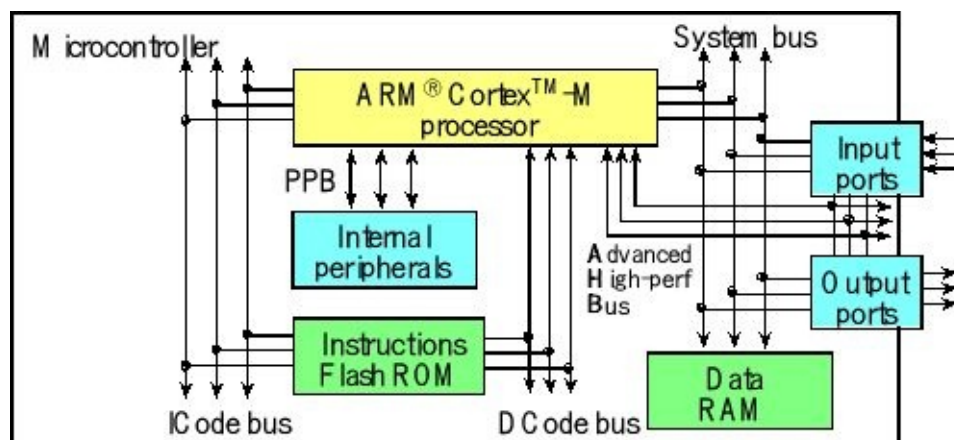


Figure 1.6. Harvard architecture of an ARM Cortex-M-based microcontroller.

The instruction set combines the high performance typical of a 32-bit processor with high code density typical of 8-bit and 16-bit microcontrollers. Instructions are fetched from flash ROM using the ICode bus. Data are exchanged with memory and

I/O via the system bus interface. There are many sophisticated debugging features utilizing the DCode bus. An **interrupt** is a hardware-triggered software function, which is extremely important for real-time embedded systems. The **latency** of an interrupt service is the time between hardware trigger and software response. Some internal peripherals, like the nested vectored interrupt controller (NVIC), communicate directly with the processor via the private peripheral bus (PPB). The tight integration of the processor and interrupt controller provides fast execution of interrupt service routines (ISRs), dramatically reducing the interrupt latency.

**Checkpoint 1.4:** Why do you suppose the Cortex M has so many busses?

**Checkpoint 1.5:** Notice the debugger exists on the DCode bus. Why is this a good idea?

## 1.2.2. Memory

One kibibyte (KiB) equals 1024 bytes of memory. The TM4C123 has 256 kibibytes ( $2^{18}$  bytes) of flash ROM and 32 kibibytes ( $2^{15}$  bytes) of RAM. The MSP432 also has 256 kibibytes ( $2^{18}$  bytes) of flash ROM but has 64 kibibytes ( $2^{16}$  bytes) of RAM. We view the memory as continuous virtual address space with the RAM beginning at 0x2000.0000, and the flash ROM beginning at 0x0000.0000.

The microcontrollers in the Cortex-M family differ by the amount of memory and by the types of I/O modules. There are hundreds of members in this family; some of them are listed in Table 1.2. The memory maps of TM4C123 and MSP432 are shown in Figure 1.7. Although this course focuses on two microcontrollers from Texas Instruments, all ARM Cortex-M microcontrollers have similar memory maps. In general, Flash ROM begins at address 0x0000.0000, RAM begins at 0x2000.0000, the peripheral I/O space is from 0x4000.0000 to 0x5FFF.FFFF, and I/O modules on the private peripheral bus exist from 0xE000.0000 to 0xE00F.FFFF. In particular, the only differences in the memory map for the various members of the Cortex-M family are the ending addresses of the flash and RAM.

<i>Part number</i>	<i>RAM</i>	<i>Flash</i>	<i>I/O</i>	<i>I/O modules</i>
MSP432P401RIPZ	64	256	84	floating point, DMA
TM4C123GH6PM	32	256	43	floating point, CAN, DMA, USB, PWM
TM4C1294NCPDT	256	1024	90	floating point, CAN, DMA, USB, PWM, Ethernet
STM32F051R8T6	8	64	55	DAC, Touch sensor, DMA, I2S, HDMI, PWM
MKE02Z64VQH2	4	64	53	PWM
	KiB	KiB	pins	

**Table 1.2. Memory and I/O modules (all have SysTick, RTC, timers, UART, I<sup>2</sup>C, SSI, and**



## ADC).

Having multiple buses means the processor can perform multiple tasks in parallel. On the TM4C123, general purpose input/output (GPIO) ports can be accessed using either the PPB or AHPB. The following is some of the tasks that can occur in parallel

ICode bus	Fetch opcode from ROM
DCode bus	Read constant data from ROM
System bus	Read/write data from RAM or I/O, fetch opcode from RAM
PPB	Read/write data from internal peripherals like the NVIC
AHPB	Read/write data from internal peripherals like the USB

Instructions and data are accessed using a common bus on a von Neumann machine. The Cortex-M processor is a Harvard architecture because instructions are fetched on the ICode bus and data accessed on the system bus. The address signals on the ARM Cortex-M processor include 32 lines, which together specify the memory address (0x0000.0000 to 0xFFFF.FFFF) that is currently being accessed. The address specifies both which module (input, output, RAM, or ROM) as well as which cell within the module will communicate with the processor. The data signals contain the information that is being transferred and also include 32 bits. However, on the system bus it can also transfer 8-bit or 16-bit data. The control signals specify the timing, the size, and the direction of the transfer.

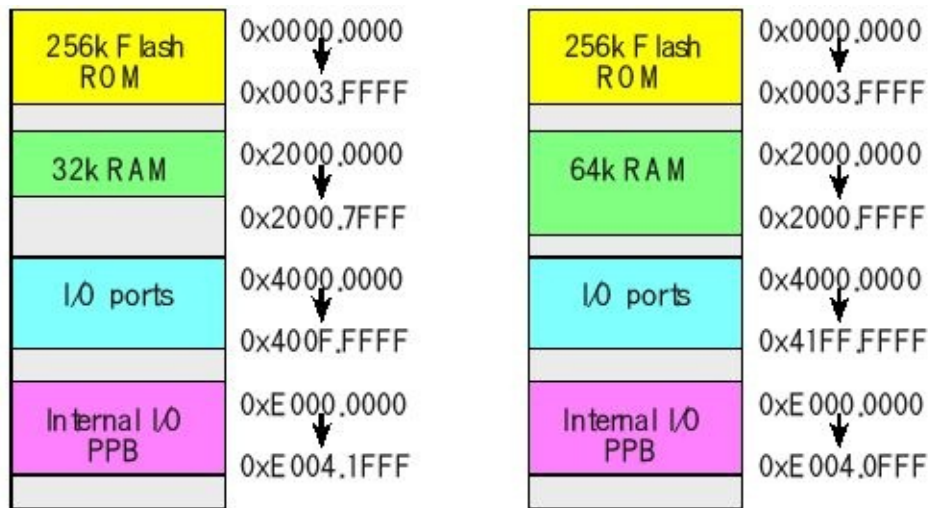


Figure 1.7. Memory map of the TM4C123 with 256k ROM and 32k RAM and the MSP432 with 256k ROM and 64k RAM.

**Checkpoint 1.6:** What do we put in RAM and what do we put in ROM?

**Checkpoint 1.7:** Can software write into the ROM of our microcontroller?

The ARM Cortex-M processor uses **bit-banding** to allow read/write access to individual bits in RAM and some bits in the I/O space. There are two parameters that define bit-banding: the address and the bit you wish to access. Assume you wish to access bit  $b$  of RAM address  $0x2000.0000+n$ , where  $b$  is a number 0 to 7. The aliased address for this bit will be

$$0x2200.0000 + 32*n + 4*b$$

Reading this address will return a 0 or a 1. Writing a 0 or 1 to this address will perform an atomic read-modify-write modification to the bit.

If we consider 32-bit word-aligned data in RAM, the same bit-banding formula still applies. Let the word address be  $0x2000.0000+n$ .  $n$  starts at 0 and increments by 4. In this case, we define  $b$  as the bit from 0 to 31. In little-endian format, bit 1 of the byte at  $0x2000.0001$  is the same as bit 9 of the word at  $0x2000.0000$ . The aliased address for this bit will still be

$$0x2200.0000 + 32*n + 4*b$$

Examples of bit-banded addressing are listed in Table 1.3. Writing a 1 to location  $0x2200.0018$  will set bit 6 of RAM location  $0x2000.0000$ . Reading location  $0x2200.0024$  will return a 0 or 1 depending on the value of bit 1 of RAM location  $0x2000.0001$ .

<i>RAM address</i>	<i>Offset n</i>	<i>Bit b</i>	<i>Bit-banded alias</i>
0x2000.0000	0	0	0x2200.0000
0x2000.0000	0	1	0x2200.0004
0x2000.0000	0	2	0x2200.0008
0x2000.0000	0	3	0x2200.000C
0x2000.0000	0	4	0x2200.0010
0x2000.0000	0	5	0x2200.0014
0x2000.0000	0	6	0x2200.0018
0x2000.0000	0	7	0x2200.001C
0x2000.0001	1	0	0x2200.0020
0x2000.0001	1	1	0x2200.0024

**Table 1.3. Examples of bit-banded addressing.**

**Checkpoint 1.8:** What address do you use to access bit 3 of the byte at  $0x2000.1010$ ?

**Checkpoint 1.9:** What address do you use to access bit 22 of the word at  $0x2001.0000$ ?

The other bit-banding region is the I/O space from  $0x4000.0000$  through  $0x400F.FFFF$ . In this region, let the I/O address be  $0x4000.0000+n$ , and let  $b$  represent the bit 0 to 7. The aliased address for this bit will be  $0x4200.0000 + 32*n + 4*b$

**Checkpoint 1.10:** What address do you use to access bit 7 of the byte at  $0x4000.0030$ ?



# 1.3. Cortex-M Processor Architecture

## 1.3.1. Registers

The **registers** on an ARM Cortex-M processor are depicted in Figure 1.8. R0 to R12 are general purpose registers and contain either data or addresses. Register R13 (also called the stack pointer, SP) points to the top element of the stack. Actually, there are two stack pointers: the main stack pointer (MSP) and the process stack pointer (PSP). Only one stack pointer is active at a time. In a high-reliability operating system, we could activate the PSP for user software and the MSP for operating system software. This way the user program could crash without disturbing the operating system. Most of the commercially available real-time operating systems available on the Cortex M will use the PSP for user code and MSP for OS code. Register R14 (also called the link register, LR) is used to store the return location for functions. The LR is also used in a special way during exceptions, such as interrupts. Register R15 (also called the program counter, PC) points to the next instruction to be fetched from memory. The processor fetches an instruction using the PC and then increments the PC by the length (in bytes) of the instruction fetched.

**Checkpoint 1.11:** How are registers R13 R14 and R15 special?

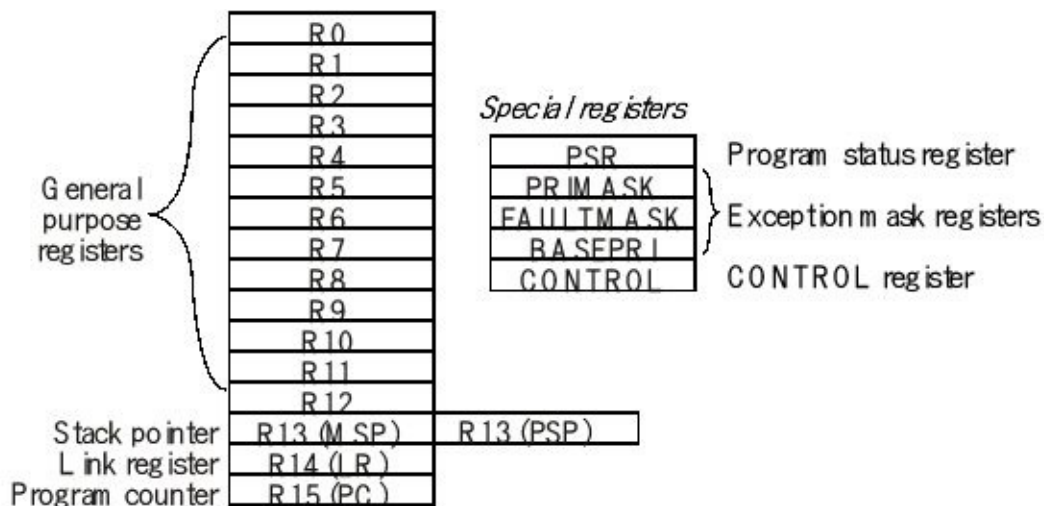


Figure 1.8. The registers on the ARM Cortex-M processor.

The **ARM Architecture Procedure Call Standard**, AAPCS, part of the **ARM Application Binary Interface (ABI)**, uses registers R0, R1, R2, and R3 to pass input parameters into a C function or an assembly subroutine. Also according to AAPCS we place the return parameter in Register R0. The standard requires functions to preserve the contents of R4-R11. In other words, functions save R4-R11, use R4-

R11, and then restore R4-R11 before returning. Another restriction is to keep the stack aligned to 64 bits, by pushing and popping an even number of registers.

There are three status registers named Application Program Status Register (APSR), the Interrupt Program Status Register (IPSR), and the Execution Program Status Register (EPSR) as shown in Figure 1.9. These registers can be accessed individually or in combination as the **Program Status Register (PSR)**.

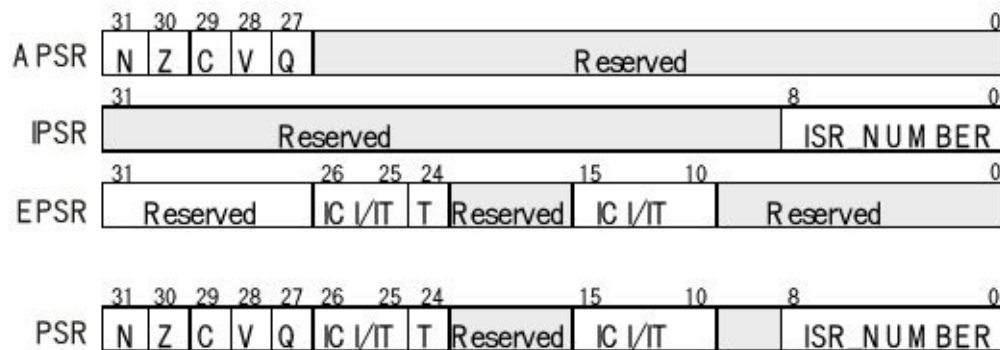


Figure 1.9. The program status register of the ARM Cortex-M processor.

The N, Z, V, C, and Q bits signify the status of the previous ALU operation. Many instructions set these bits to signify the result of the operation. In general, the **N bit** is set after an arithmetical or logical operation signifying whether or not the result is negative. Similarly, the **Z bit** is set if the result is zero. The **C bit** means carry and is set on an unsigned overflow, and the **V bit** signifies signed overflow. The **Q bit** is the sticky saturation flag, indicating that “saturation” has occurred, and is set by the **SSAT** and **USAT** instructions.

The **T bit** will always be 1, indicating the ARM Cortex-M processor is executing Thumb instructions. The IC/IT bits are used by interrupts and by IF-THEN instructions. The ISR\_NUMBER indicates which interrupt if any the processor is handling. Bit 0 of the special register **PRIMASK** is the interrupt mask bit, or **I bit**. If this bit is 1 most interrupts and exceptions are not allowed. If the bit is 0, then interrupts are allowed. Bit 0 of the special register **FAULTMASK** is the fault mask bit. If this bit is 1 all interrupts and faults are disallowed. If the bit is 0, then interrupts and faults are allowed. The nonmaskable interrupt (NMI) is not affected by these mask bits. The **BASEPRI** register defines the priority of the executing software. It prevents interrupts with lower or equal priority from interrupting the current execution but allows higher priority interrupts. For example if **BASEPRI** equals 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. The details of interrupt processing will be presented in detail, later in the book.

**Checkpoint 1.12:** Where is the I bit and what does it mean?

## 1.3.2. Stack

The **stack** is a last-in-first-out temporary storage. Managing the stack is an important function for the operating system. To create a stack, a block of RAM is allocated for this temporary storage. On the ARM Cortex-M processor, the stack always operates on 32-bit data. The stack pointer (SP) points to the 32-bit data on the top of the stack. The stack grows downwards in memory as we push data on to it so, although we refer to the most recent item as the “top of the stack” it is actually the item stored at the lowest address! To **push** data on the stack, the stack pointer is first decremented by 4, and then the 32-bit information is stored at the address specified by SP. To **pop** data from the stack, the 32-bit information pointed to by SP is first retrieved, and then the stack pointer is incremented by 4. SP points to the last item pushed, which will also be the next item to be popped. The processor allows for two stacks, the main stack and the process stack, with independent copies of the stack pointer. The boxes in Figure 1.10 represent 32-bit storage elements in RAM. The grey boxes in the figure refer to actual data stored on the stack, and the white boxes refer to locations in memory that do not contain stack data. This figure illustrates how the stack is used to push the contents of Registers R0, R1, and R2 in that order. Assume Register R0 initially contains the value 1, R1 contains 2 and R2 contains 3. The drawing on the left shows the initial stack. The software executes these six

```

PUSH {R0}
PUSH {R1}
PUSH {R2}
POP {R3}
POP {R4}
POP {R5}

```

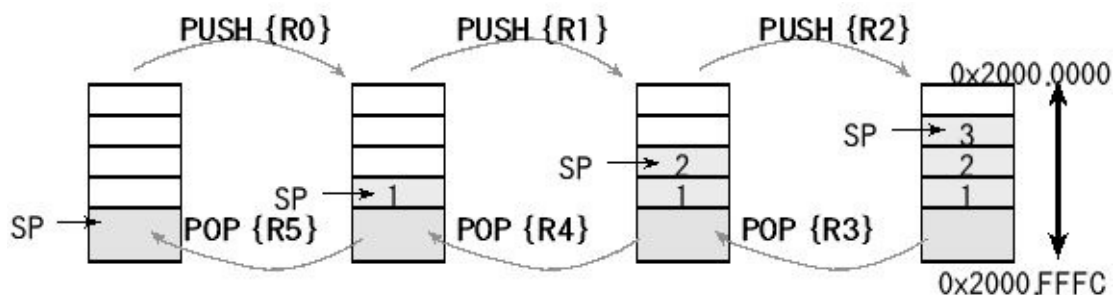


Figure 1.10. Stack picture showing three numbers first being pushed, then three numbers being popped.

We can push and pop multiple registers; these six instructions could be replaced with

```

PUSH {R0-R2}
POP {R3-R5}

```

The instruction **PUSH {R0}** saves the value of R0 on the stack. It first decrements SP by 4, and then it stores the contents of R0 into the memory location pointed to by SP. The right-most drawing shows the stack after the push occurs three times. The stack contains the numbers 1 2 and 3, with 3 on top. The instruction **POP{R3}** retrieves

data from the stack. It first moves the value from memory pointed to by SP into R3, and then it increments SP by 4. After the pop occurs three times the stack reverts to its original state and registers R3, R4 and R5 contain 3 2 1 respectively. We define the 32-bit word pointed to by SP as the **top** entry of the stack. If it exists, we define the 32-bit data immediately below the top, at SP+4, as **next** to top. Proper use of the stack requires following these important rules

1. Functions should have an equal number of pushes and pops
2. Stack accesses (push or pop) should not be performed outside the allocated area
3. Stack reads and writes should not be performed within the free area
4. Stack push should first decrement SP, then store the data
5. Stack pop should first read the data, and then increment SP

Functions that violate rule number 1 will probably crash when incorrect data are popped off at a later time. Violations of rule number 2 can be caused by a stack underflow or overflow. Overflow occurs when the number of elements became larger than the allocated space. Stack underflow is caused when there are more pops than pushes, and is always the result of a software bug. A stack overflow can be caused by two reasons. If the software mistakenly pushes more than it pops, then the stack pointer will eventually overflow its bounds. Even when there is exactly one pop for each push, a stack overflow can occur if the stack is not allocated large enough. The processor will generate a **bus fault** when the software tries read from or write to an address that doesn't exist. If valid RAM exists below the stack then further stack operations will corrupt data in this memory.

First, we will consider the situation where the allocated stack area is placed at the beginning of RAM. For example, assume we allocate 4096 bytes for the stack from 0x2000.0000 to 0x2000.0FFF; see the left side of Figure 1.11. The SP is initialized to 0x2000.1000, and the stack is considered empty. If the SP becomes less than 0x2000.0000 a **stack overflow** has occurred. The stack overflow will cause a bus fault because there is nothing at address 0x1FFF.FFFC. If the software tries to read from or write to any location greater than or equal to 0x2000.1000 then a **stack underflow** has occurred. At this point the stack and global variables exist at overlapping addresses. Stack underflow is a very difficult bug to recognize, because the first consequence will be unexplained changes to data stored in global variables.

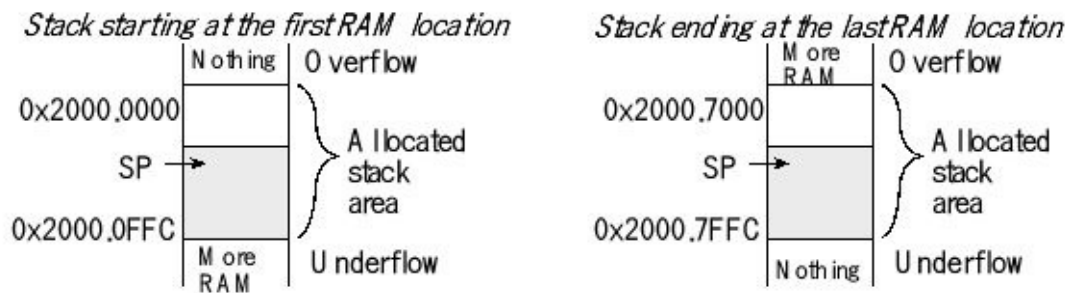


Figure 1.11. Drawings showing two possible ways to allocate the stack area in RAM.

Next, we will consider the situation where the allocated stack area is placed at the end of RAM. The TM4C123 has 32 KiB of RAM from 0x2000.0000 to 0x2000.7FFF. So in this case we allocate the 4096 bytes for the stack from 0x2000.7000 to 0x2000.7FFF, shown on the right side of Figure 1.11. The SP is initialized to 0x2000.8000, and the stack is considered empty. If the SP becomes less than 0x2000.7000 a stack overflow has occurred. The stack overflow will not cause a bus fault because there is memory at address 0x2000.6FFC. Stack overflow in this case is a very difficult bug to recognize, because the first consequence will be unexplained changes to data stored below the stack region. If the software tries to read from or write to any location greater than or equal to 0x2000.8000 then a stack underflow has occurred. In this case, stack underflow will cause a bus fault.

Executing an interrupt service routine will automatically push eight 32-bit words onto the stack. Since interrupts are triggered by hardware events, exactly when they occur is not under software control. Therefore, violations of rules 3, 4, and 5 will cause erratic behavior when operating with interrupts. Rules 4 and 5 are followed automatically by the **PUSH** and **POP** instructions.

### 1.3.3. Operating modes

The ARM Cortex-M processor has two privilege levels called privileged and unprivileged. Bit 0 of the **CONTROL** register is the **thread mode privilege level** (TPL). If TPL is 1 the processor level is privileged. If the bit is 0, then processor level is unprivileged. Running at the unprivileged level prevents access to various features, including the system timer and the interrupt controller. Bit 1 of the **CONTROL** register is the active stack pointer selection (ASPSEL). If ASPSEL is 1, the processor uses the PSP for its stack pointer. If ASPSEL is 0, the MSP is used. When designing a high-reliability operating system, we will run the user code at an unprivileged level using the PSP and the OS code at the privileged level using the MSP.

The processor knows whether it is running in the foreground (i.e., the main program) or in the background (i.e., an interrupt service routine). ARM defines the foreground as **thread mode**, and the background as **handler mode**. Switching between thread

and handler modes occurs automatically. The processor begins in thread mode, signified by `ISR_NUMBER=0`. Whenever it is servicing an interrupt it switches to handler mode, signified by setting `ISR_NUMBER` to specify which interrupt is being processed. All interrupt service routines run using the MSP. In particular, the context is saved onto whichever stack pointer is active, but during the execution of the ISR, the MSP is used. For a high reliability operation all interrupt service routines will reside in the operating system. User code can be run under interrupt control by providing hooks, which are function pointers. The user can set function pointers during initialization, and the operating system will call the function during the interrupt service routine.

**Observation:** Processor modes and the stack are essential components of building a reliable operating system. In particular the processor mode is an architectural feature that allows the operating system to restrict access to critical system resources.

### 1.3.4. Reset

A reset occurs immediately after power is applied and can also occur by pushing the reset button available on most boards. After a reset, the processor is in thread mode, running at a privileged level, and using the MSP stack pointer. The 32-bit value at flash ROM location 0 is loaded into the SP. All stack accesses are word aligned. Thus, the least significant two bits of SP must be 0. A reset also loads the 32-bit value at location 4 into the PC. This value is called the reset vector. All instructions are halfword aligned. Thus, the least significant bit of PC must be 0. However, the assembler will set the least significant bit in the reset vector, so the processor will properly initialize the Thumb bit (T) in the PSR. On the Cortex-M processor, the T bit should always be set to 1. On reset, the processor initializes the LR to `0xFFFFFFFF`.

### 1.3.5. Clock system

Normally, the execution speed of a microcontroller is determined by an external crystal. The Texas Instruments MSP-EXP432P401R board has a 48 MHz crystal. The Texas Instruments EK-TM4C123GXL and EK-TM4C1294-XL boards have a 16 MHz crystal. The TM4C microcontrollers have a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the bus clock will require less power to operate and generate less heat. Speeding up the bus clock obviously allows for more calculations per second.

The default bus speed of the MSP432 and TM4C microcontrollers is that of the

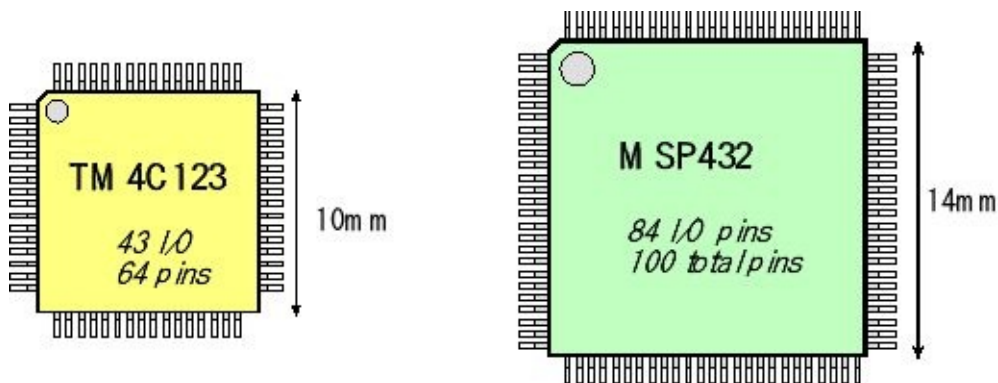
internal oscillator. For example, the default bus speed for the MSP432 is 3 MHz  $\pm 0.5\%$ . The default bus speed for the TM4C internal oscillator is 16 MHz  $\pm 1\%$ . The internal oscillator is significantly less precise than the crystal, but it requires less power and does not need an external crystal. This means for most applications we will activate the main oscillator using the crystal so we can have a stable bus clock. We will call library functions to select the clock source and bus frequency. In this book, we will assume the MSP432 is running at 48 MHz, the TM4C123 is running at 80 MHz, and the TM4C1294 is running at 120 MHz. For more details on the clock systems refer to Volume 2 of this series.

## 1.4. Texas Instruments Cortex-M Microcontrollers

### 1.4.1. Introduction to I/O

I/O is an important part of embedded systems in general. One of the important features of an operating system is to manage I/O. Input and output are the means of an embedded system to interact with its world. The external devices attached to the microcontroller provide functionality for the system. These devices connect to the microcontroller through ports. A **pin** is a specific wire on the microcontroller through which we perform input or output. A collection of pins grouped by common functionality is called a **port**. An **input port** is hardware on the microcontroller that allows information about the external world to enter into the computer. The microcontroller also has hardware called an **output port** to send information out to the external world. The GPIO (General Purpose Input Output) pins on a microcontroller are programmable to be digital input, digital output, analog input or complex and protocol (like UART etc.) specific.

Microcontrollers use most of their pins for I/O (called GPIO), see Figure 1.12. Only a few pins are not used for I/O. Examples of pins not used for I/O include power, ground, reset, debugging, and the clock. More specifically, the TM4C123 uses 43 of its 64 pins for I/O. The TM4C1294 uses 90 of its 128 pins for I/O. Similarly, the MSP432 uses 84 of its 100 pins for I/O.



*Figure 1.12. Most of the pins on the microcontroller can perform input/output.*

An **interface** is defined as the collection of the I/O port, external electronics, physical devices, and the software, which combine to allow the computer to communicate with the external world. An example of an input interface is a switch, where the operator toggles the switch, and the software can recognize the switch position. An example of an output interface is a light-emitting diode (LED), where the software can turn the light on and off, and the operator can see whether or not the light is shining. There is a wide range of possible inputs and outputs, which can exist



in either digital or analog form. In general, we can classify I/O interfaces into four categories

**Parallel/Digital** - binary data are available simultaneously on a group of lines

**Serial** - binary data are available one bit at a time on a single line

**Analog** - data are encoded as an electrical voltage, current or power

**Time** - data are encoded as a period, frequency, pulse width or phase shift

In a system with **memory-mapped I/O**, as shown in Figure 1.13, the I/O ports are connected to the processor in a manner similar to memory. I/O ports are assigned addresses, and the software accesses I/O using reads and writes to the specific I/O addresses. These addresses appear like regular memory addresses, except accessing them results in manipulation of a functionality of the mapped I/O port, hence the term memory-mapped I/O. As a result, the software inputs from an input port using the same instructions as it would if it were reading from memory. Similarly, the software outputs from an output port using the same instructions as it would if it were writing to memory.

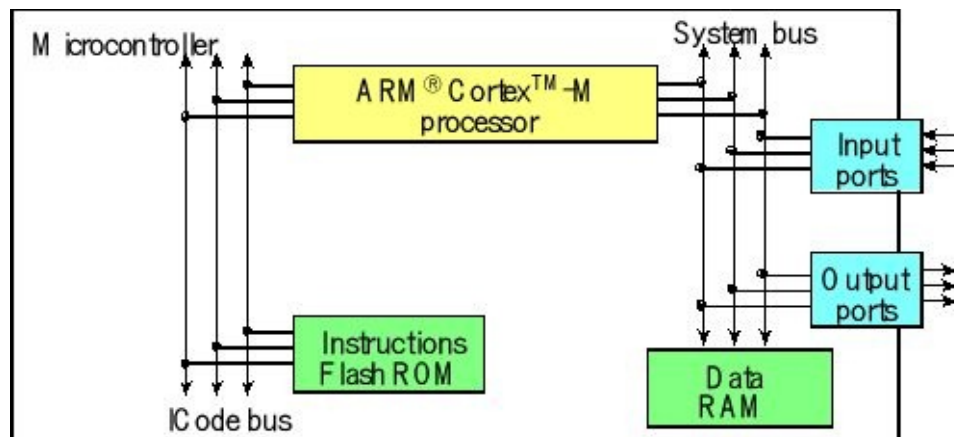


Figure 1.13. Memory-mapped input/output.

Most pins on Cortex M microcontrollers can be used for **general purpose I/O** (GPIO) called regular functions or for more complex functions called alternate functions. For example, port pins PA1 and PA0 on the TM4C123 can be either regular parallel port pins, or an asynchronous serial port called universal asynchronous receiver/transmitter (UART).

Some of the alternative functions used in this book are:

- **UART**                      **Universal asynchronous receiver/transmitter**
- **SSI or SPI**                **Synchronous serial interface or serial peripheral**

## interface

- **I<sup>2</sup>C**                    **Inter-integrated circuit**
- **Timer**                    **Periodic interrupts**
- **PWM**                    **Pulse width modulation**
- **ADC**                    **Analog to digital converter, measurement analog signals**

The **UART** can be used for serial communication between computers. It is asynchronous and allows for simultaneous communication in both directions. The **SSI** (also called **SPI**) is used to interface medium-speed I/O devices. In this class, we will use SSI to interface a graphics display. **I<sup>2</sup>C** is a simple I/O bus that we will use to interface low speed peripheral devices. In this class we use I<sup>2</sup>C to interface a light sensor and a temperature sensor. We will use the timer modules to create periodic interrupts. **PWM** outputs could be used to apply variable power to motor interfaces. However, in this class we use PWM to adjust the volume of the buzzer. The **ADC** will be used to measure the amplitude of analog signals, and will be important in data acquisition systems. In this class we will connect the microphone, joystick and accelerometer to the ADC.

Joint Test Action Group (**JTAG**), standardized as the IEEE 1149.1, is a standard test access port used to program and debug the microcontroller board. Each microcontroller uses four port pins for the JTAG interface.

**Checkpoint 1.13:** What is the difference between a pin and a port?

**Checkpoint 1.14:** List four types of input/output.

### 1.4.2. Texas Instruments TM4C123 LaunchPad I/O pins

Figure 1.14 draws the I/O port structure for the TM4C123GH6PM. This microcontroller is used on the EK-TM4C123GXL LaunchPad. Pins on the TM4C family can be assigned to as many as eight different I/O functions. Pins can be configured for digital I/O, analog input, timer I/O, or serial I/O. For example PB4 can be a digital I/O, ADC, SSI, PWM, timer or CAN pin. There are two buses used for I/O. The digital I/O ports are connected to both the advanced peripheral bus and the advanced high-performance bus (runs faster). Because of the multiple buses, the microcontroller can perform I/O bus cycles simultaneous with instruction fetches from flash ROM. The TM4C123GH6PM has eight UART ports, four SSI ports, four I2C ports, two 12-bit ADCs, twelve timers, two PWMs, a CAN port, and a USB interface. There are 43 I/O lines. There are twelve ADC inputs; each ADC can convert up to 1M samples per second. Table 1.4 lists the regular and alternate names of the port pins.

Each pin has one configuration bit in the GPIOAMSEL register. We set this bit to connect the port pin to the ADC or analog comparator. For digital functions, each pin

also has four bits in the GPIOCTL register, which we set to specify the alternative function for that pin (0 means regular I/O port). Not every pin can be connected to every alternative function. See Table 1.4.

Pins PC3 – PC0 were left off Table 1.4 because these four pins are reserved for the JTAG debugger, and should not be used for regular I/O. Notice, most alternate function modules (e.g., UORx) only exist on one pin (PA0). While other functions could be mapped to two or three pins (CAN0Rx could be mapped to PB4, PE4 or PF3.)

The two pins PD7 and PF0 are associated with NMI; these two pins are initially locked. This means if you plan to use PD7 or PF0 you will need to unlock it by first writing 0x4C4F434B to the lock register and then setting bits in the commit register. This code unlocks PF0

```
GPIO_PORTF_LOCK_R = 0x4C4F434B; // unlock GPIO Port F
GPIO_PORTF_CR_R = 0x1F; // allow changes to PF4-0
```

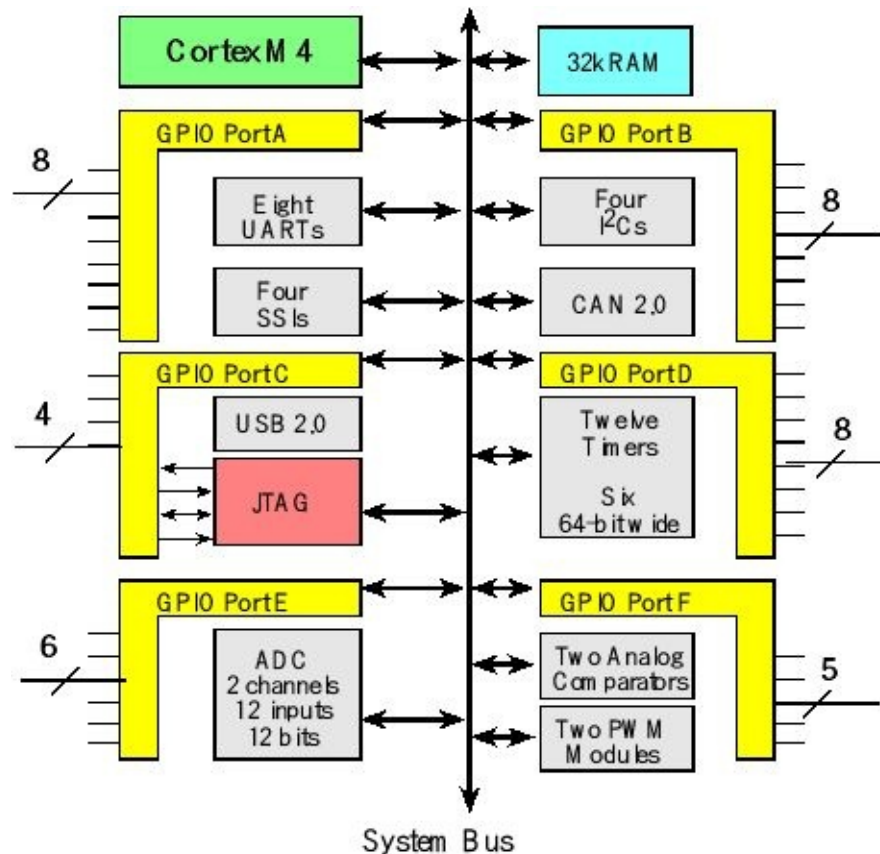


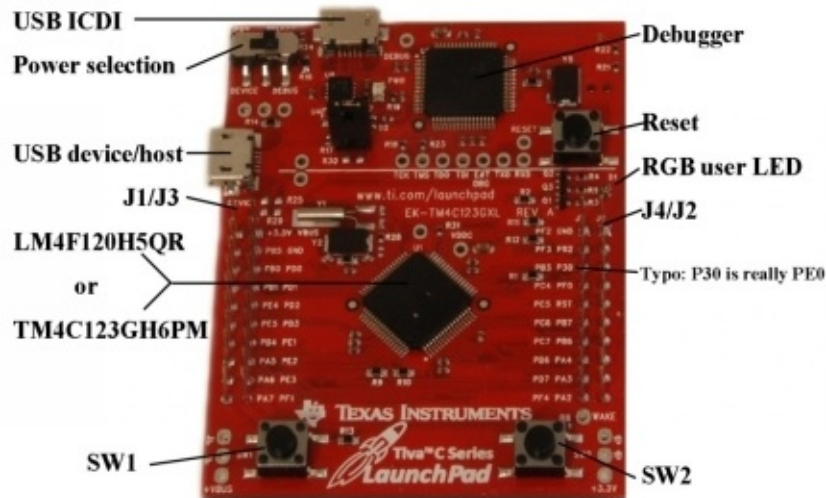
Figure 1.14. I/O port pins for the TM4C123GH6PM microcontroller.

For example, if we wished to use UART7 on pins PE0 and PE1, we would set bits 1,0 in the digital enable register (enable digital), clear bits 1,0 in the **GPIO\_PORTE\_AMSEL\_R** register (disable analog) and set the PMCx bits in the for PE0 PE1 to 0001 (enable alternate function) in the **GPIO\_PORTE\_PCTL\_R** register. If we wished to sample an analog signal on PD0, we would clear bit 0 in the digital enable register (disable digital), and set bit



PE3	Ain0	Port										
PE4	Ain9	Port	U5Rx		I <sub>2</sub> C2SCL	M0PWM4	M1PWM2			CAN0Rx		
PE5	Ain8	Port	U5Tx		I <sub>2</sub> C2SDA	M0PWM5	M1PWM3			CAN0Tx		
PF0		Port	U1RTS	SSI1Rx	CAN0Rx		M1PWM4	PhA0	T0CCP0	NMI	C0o	
PF1		Port	U1CTS	SSI1Tx			M1PWM5	PhB0	T0CCP1		C1o	TRD1
PF2		Port		SSI1Clk		M0Fault0	M1PWM6		T1CCP0			TRD0
PF3		Port		SSI1Fss	CAN0Tx		M1PWM7		T1CCP1			TRCLK
PF4		Port					M1Fault0	IDX0	T2CCP0	USB0open		

**Table 1.4. PMCx bits in the GPIOCTL register on the LM4F/TM4C specify alternate functions. PB1, PB0, PD4 and PD5 are hardwired to the USB device. PA0 and PA1 are hardwired to the serial port. PWM is not available on LM4F120.**



*Figure 1.15. Tiva TM4C123 Launchpad Evaluation Board based on the TM4C123GH6PM.*

Pins PA1 – PA0 create a serial port, which is linked through the debugger cable to the PC. The serial link is a physical UART as seen by the TM4C and mapped to a virtual COM port on the PC. The USB device interface uses PD4 and PD5. The JTAG debugger requires pins PC3 – PC0. The LaunchPad connects PB6 to PD0, and PB7 to PD1. If you wish to use both PB6 and PD0 you will need to remove the R9 resistor. Similarly, to use both PB7 and PD1 remove the R10 resistor.

The TM4C123 LaunchPad evaluation board has two switches and one 3-color LED. See Figure 1.16. The switches are negative logic and will require activation of the internal pull-up resistors. In particular, you will set bits 0 and 4 in **GPIO\_PORTF\_PUR\_R** register. The LED interfaces on PF3 – PF1 are positive logic. To use the LED, make the PF3 – PF1 pins an output. To activate the red color, output a one to PF1. The blue color is on PF2, and the green color is controlled by PF3. The 0-Ω resistors (R1, R2, R11, R12, and R13) can be removed to disconnect the corresponding pin from the external hardware.

The LaunchPad has four 10-pin connectors, labeled as J1 J2 J3 J4 in Figures 1.15 and 1.17, to which you can attach your external signals. The top side of these connectors has male pins and the bottom side has female sockets. The intent is to stack boards together to make a layered system see Figure 1.17. Texas Instruments

also supplies Booster Packs, which are pre-made external devices that will plug into this 40-pin connector. The Booster Packs for the MSP430 LaunchPad are compatible (one simply plugs these 20-pin connectors into the outer two rows) with this board. The inner 10-pin headers (connectors J3 and J4) are not intended to be compatible with other TI LaunchPads. J3 and J4 apply only to Tiva Booster Packs.

There are a number of good methods to connect external circuits to the LaunchPad. One method is to purchase a male to female jumper cable (e.g., item number 826 at [www.adafruit.com](http://www.adafruit.com)). A second method is to solder a solid wire into a female socket (e.g., Hirose DF11-2428SCA) creating a male to female jumper wire.

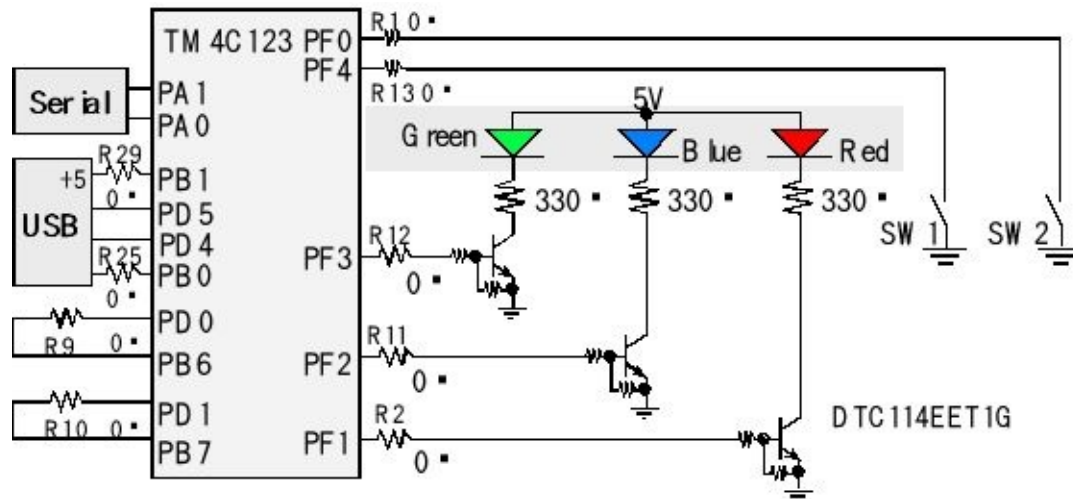


Figure 1.16. Switch and LED interfaces on the Texas Instruments TM4C123 LaunchPad Evaluation Board. The zero ohm resistors can be removed so the corresponding pin can be used for its regular purpose.

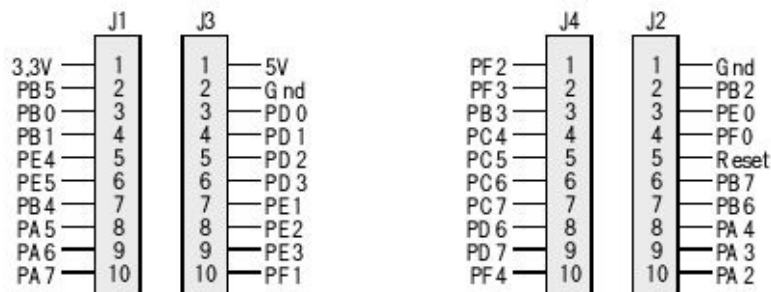


Figure 1.17. Interface connectors on the Texas Instruments TM4C123 LaunchPad Evaluation Board.

### 1.4.3. Texas Instruments TM4C1294 Connected LaunchPad I/O pins

Figure 1.18 shows the 90 I/O pins available on the TM4C1294NCPDT, which is the microcontroller used on the Connected LaunchPad. Pins on the TM4C family can be assigned to as many as seven different I/O functions, see Table 1.5. Pins can be configured for digital I/O, analog input, timer I/O, or serial I/O. For example PA0 can

be digital I/O, serial input, I2C clock, Timer I/O, or CAN receiver. There are two buses used for I/O. Unlike the TM4C123, the digital I/O ports are only connected to the advanced high-performance bus. The microcontroller can perform I/O bus cycles simultaneous with instruction fetches from flash ROM. The TM4C1294NCPDT has eight UART ports, four SSI ports, ten I2C ports, two 12-bit ADCs, eight timers, two CAN ports, a USB interface, 8 PWM outputs, and an Ethernet port. Of the 90 I/O lines, twenty pins can be used for analog inputs to the ADC. The ADC can convert up to 1M samples per second. Table 1.5 lists the regular and alternate functions of the port pins.

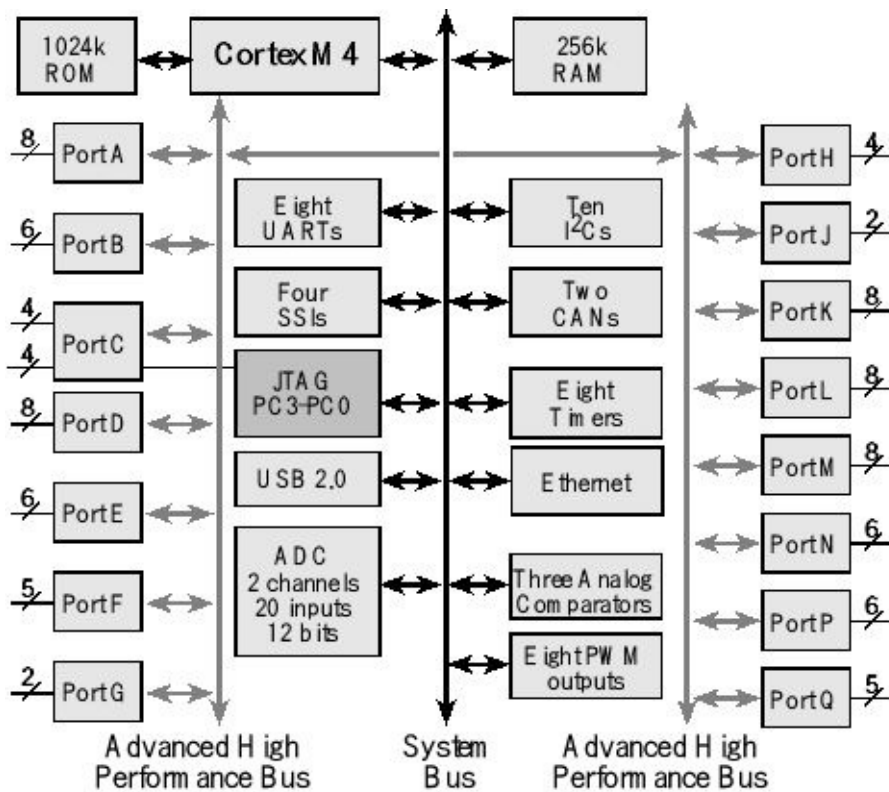


Figure 1.18. I/O port pins for the TM4C1294NCPDT microcontroller.

Figure 1.19 shows the pin locations of the two Booster Pack connectors. There are three methods to connect external circuits to the Connected LaunchPad. One method uses male to female jumper cable (e.g., item number 826 at [www.adafruit.com](http://www.adafruit.com)) or solder a solid wire into a female socket (e.g., Hirose DF11-2428SCA) creating a male-to-female jumper wire. In this method, you connect the female socket to the top of the LaunchPad and the male pin into a solderless breadboard. The second method uses male-to-male wires interfacing to the bottom of the LaunchPad. The third method uses two 49-pin right-angle headers so the entire LaunchPad can be plugged into a breadboard. You will need one each of Samtec parts TSW-149-09-L-S-RE and TSW-149-08-L-S-RA. This configuration is shown in Figure 1.20, and directions can be found at <http://users.ece.utexas.edu/~valvano/arm/TM4C1294soldering.pdf>

The Connected LaunchPad has two switches and four LEDs. Switch SW1 is connected to pin PJ0, and SW2 is connected to PJ1. These two switches are negative logic and require enabling the internal pull up (**PUR**). A reset switch will reset the



microcontroller and your software will start when you release the switch. Positive logic LEDs D1, D2, D3, and D4 are connected to PN1, PN0, PF4, and PF0 respectively. A power LED indicates that 3.3 volt power is present on the board. R19 is a 0  $\Omega$  resistor connecting PA3 and PQ2. Similarly, R20 is a 0  $\Omega$  resistor connecting PA2 and PQ3. You need to remove R19 if you plan to use both PA3 and PQ2. You need to remove R20 if you plan to use both PA2 and PQ3. See Figures 1.20 and 1.21.

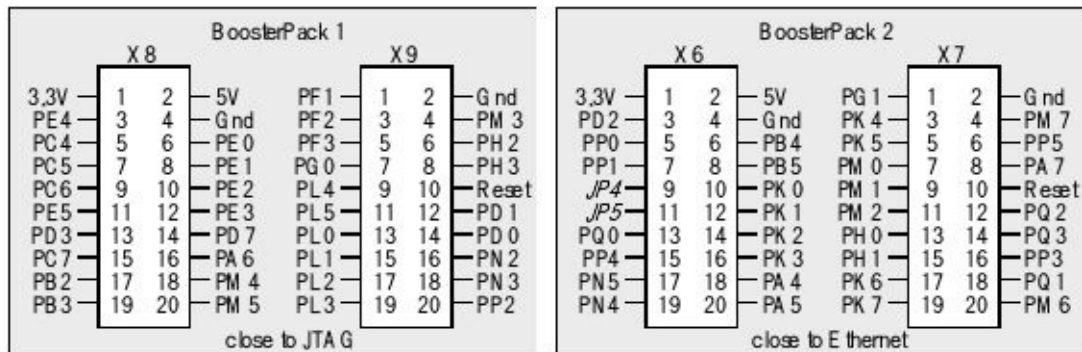


Figure 1.19. Interface connectors on the EK-TM4C1294-XL LaunchPad Evaluation Board.

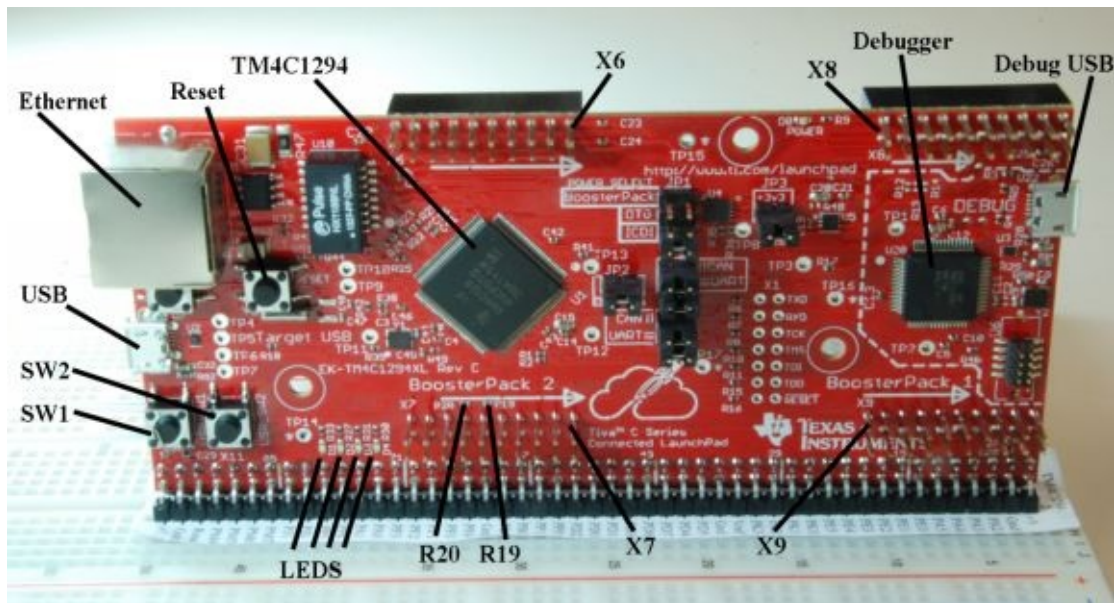


Figure 1.20. EK-TM4C1294-XL Connected LaunchPad.

Jumper JP1 has six pins creating three rows of two. Exactly one jumper should be connected in the JP1 block, which selects the power source. The top position is for BoosterPack power. The middle position draws power from the USB connector, labeled OTG, on the left side of the board near the Ethernet jack. We recommend placing the JP1 jump in the bottom position so power is drawn from the ICDI (Debug) USB connection. Under normal conditions, you should place jumpers in both J2 and J3. Jumpers J2 and J3 facilitate measuring current to the microcontroller. We recommend you place JP4 and JP5 in the “UART” position so PA1 and PA0 are connected to the PC as a virtual COM port. Your code runs on the 128-pin TM4C1294 microcontroller. There is a second TM4C microcontroller on the board,





PK1	AIN17	U4Tx	-	-	-	-	-	-	-	-	-	EPIO
PK2	AIN18	U4RTS	-	-	-	-	-	-	-	-	-	EPIO
PK3	AIN19	U4CTS	-	-	-	-	-	-	-	-	-	EPIO
PK4	-	-	I2C3SCL	-	EN0LED0	M0PWM6	-	-	-	-	-	EPIO
PK5	-	-	I2C3SDA	-	EN0LED2	M0PWM7	-	-	-	-	-	EPIO
PK6	-	-	I2C4SCL	-	EN0LED1	M0FAULT1	-	-	-	-	-	EPIO
PK7	-	U0RI	I2C4SDA	-	RTCCLK	M0FAULT2	-	-	-	-	-	EPIO
PL0	-	-	I2C2SDA	-	-	M0FAULT3	-	-	-	-	USB0D0	EPIO
PL1	-	-	I2C2SCL	-	-	PhA0	-	-	-	-	USB0D1	EPIO
PL2	-	-	-	-	C0o	PhB0	-	-	-	-	USB0D2	EPIO
PL3	-	-	-	-	C1o	IDX0	-	-	-	-	USB0D3	EPIO
PL4	-	-	-	T0CCP0	-	-	-	-	-	-	USB0D4	EPIO
Pin	Analog	1	2	3	5	6	7	11	13	14	15	
PL5	-	-	-	T0CCP1	-	-	-	-	-	-	USB0D5	EPIO
PL6	USB0DP	-	-	T1CCP0	-	-	-	-	-	-	-	
PL7	USB0DM	-	-	T1CCP1	-	-	-	-	-	-	-	
PM0	-	-	-	T2CCP0	-	-	-	-	-	-	-	EPIO
PM1	-	-	-	T2CCP1	-	-	-	-	-	-	-	EPIO
PM2	-	-	-	T3CCP0	-	-	-	-	-	-	-	EPIO
PM3	-	-	-	T3CCP1	-	-	-	-	-	-	-	EPIO
PM4	TMPR3	U0CTS	-	T4CCP0	-	-	-	-	-	-	-	
PM5	TMPR2	U0DCD	-	T4CCP1	-	-	-	-	-	-	-	
PM6	TMPR1	U0DSR	-	T5CCP0	-	-	-	-	-	-	-	
PM7	TMPR0	U0RI	-	T5CCP1	-	-	-	-	-	-	-	
PN0	-	U1RTS	-	-	-	-	-	-	-	-	-	
PN1	-	U1CTS	-	-	-	-	-	-	-	-	-	
PN2	-	U1DCD	U2RTS	-	-	-	-	-	-	-	-	EPIO
PN3	-	U1DSR	U2CTS	-	-	-	-	-	-	-	-	EPIO
PN4	-	U1DTR	U3RTS	I2C2SDA	-	-	-	-	-	-	-	EPIO
PN5	-	U1RI	U3CTS	I2C2SCL	-	-	-	-	-	-	-	EPIO
PP0	C2+	U6Rx	-	-	-	-	-	-	-	-	-	SSI3
PP1	C2-	U6Tx	-	-	-	-	-	-	-	-	-	SSI3
PP2	-	U0DTR	-	-	-	-	-	-	-	-	USB0NXT	EPIO
PP3	-	U1CTS	U0DCD	-	-	-	RTCCLK	-	-	-	USB0DIR	EPIO
PP4	-	U3RTS	U0DSR	-	-	-	-	-	-	-	USB0D7	-
PP5	-	U3CTS	I2C2SCL	-	-	-	-	-	-	-	USB0D6	-
PQ0	-	-	-	-	-	-	-	-	-	-	SSI3Clk	EPIO
PQ1	-	-	-	-	-	-	-	-	-	-	SSI3Fss	EPIO
PQ2	-	-	-	-	-	-	-	-	-	-	SSI3XDAT0	EPIO
PQ3	-	-	-	-	-	-	-	-	-	-	SSI3XDAT1	EPIO
PQ4	-	U1Rx	-	-	-	-	DIVSCLK	-	-	-	-	

**Table 1.5. PMCx bits in the GPIO\_PORTx\_PCTL\_R register on the TM4C1294 specify alternate functions. PD7 can be NMI by setting PCTL bits 31-28 to 8. PL6 and PL7 are hardwired to the USB.**

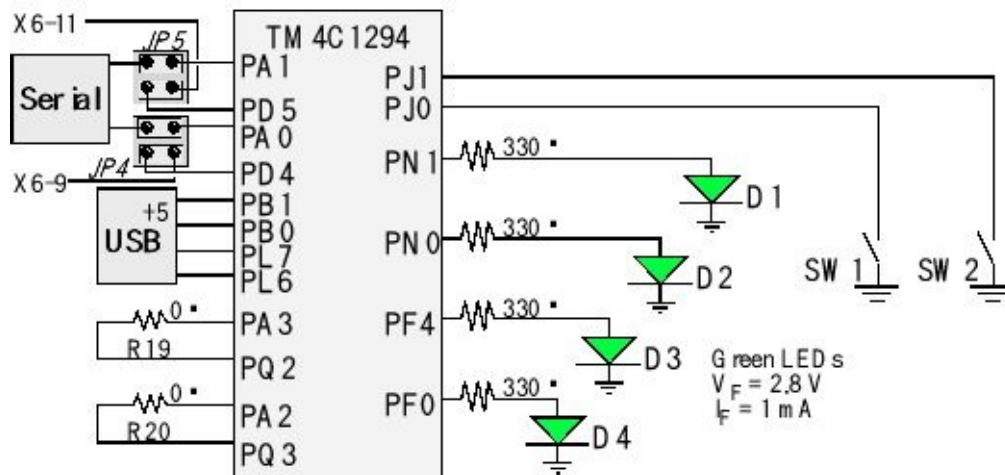
Each pin has one configuration bit in the **AMSEL** register. We set this bit to connect the port pin to the ADC or analog comparator. For digital functions, each pin also has four bits in the **PCTL** register, which we set to specify the alternative function for that pin (0 means regular I/O port). Table 1.5 shows the 4-bit **PCTL** configuration used to connect each pin to its alternate function. For example, column “3” means set

4-bit field in **PCTL** to 0011.

Pins PC3 – PC0 were left off Table 1.5 because these four pins are reserved for the JTAG debugger and should not be used for regular I/O. Notice, some alternate function modules (e.g., UORx) only exist on one pin (PA0), while other functions could be mapped to two or three pins. For example, TOCCP0 could be mapped to one of the following: PA0, PD0, or PL4.

The PCTL bits in Table 1.5 can be tricky to understand. For example, if we wished to use UART6 on pins PP0 and PP1, we would set bits 1,0 in the **DEN** register (enable), clear bits 1,0 in the **AMSEL** register (disable), write a 0001,0001 to bits 7–0 in the **PCTL** register (UART) **GPIO\_PORTP\_PCTL\_R = (GPIO\_PORTP\_PCTL\_R&0xFFFFFFFF)+0x00000011**; and set bits 1,0 in the **AFSEL** register (enable alternate function). If we wished to sample an analog signal on PD0, we would set bit 0 in the alternate function select register **AFSEL**, clear bit 0 in the digital enable register **DEN** (disable digital), set bit 0 in the analog mode select register **AMSEL** (enable analog), and activate one of the ADCs to sample channel 15.

Jumpers JP4 and JP5 select whether the serial port on UART0 (PA1 – PA0) or on UART2 (PD5 – 4) is linked through the debugger cable to the PC. The serial link is a physical UART as seen by the TM4C1294 and is mapped to a virtual COM port on the PC. The USB device interface uses PL6 and PL7. The JTAG debugger requires pins PC3 – PC0.



*Figure 1.21. Switch and LED interfaces on the Connected LaunchPad Evaluation Board. The zero ohm resistors can be removed so all the pins can be used. See Chapter 9 for Ethernet connections.*

To use the negative logic switches, make the pins digital inputs, and activate the internal pull-up resistors. In particular, you will activate the Port J clock, clear bits 0 and 1 in **GPIO\_PORTJ\_DIR\_R** register, set bits 0 and 1 in **GPIO\_PORTJ\_DEN\_R** register, and set bits 0 and 1 in **GPIO\_PORTJ\_PUR\_R** register. The LED interfaces are positive logic. To use the LEDs, make the PN1, PN0, PF4, and PF0 pins an output. You will activate the

Port N clock, set bits 0 and 1 in `GPIO_PORTN_DIR_R` register, and set bits 0 and 1 in `GPIO_PORTN_DEN_R` register. You will activate the Port F clock, set bits 0 and 4 in `GPIO_PORTF_DIR_R` register, and set bits 0 and 4 in `GPIO_PORTF_DEN_R` register.

## 1.4.4. Texas Instruments MSP432 LaunchPad I/O pins

Figure 1.22 draws the I/O port structure for the MSP432P401R. This microcontroller is used on the MSP-EXP432P401R LaunchPad. Pins can be configured for digital I/O, analog input, timer I/O, or serial I/O. For example P1.2 can be digital I/O or serial receive input.

Because of the multiple buses, the microcontroller can perform I/O bus cycles simultaneous with instruction fetches from flash ROM. The MSP432P401R has four UART ports, eight SPI ports, four I2C ports, a 14-bit ADC, and four timers. There are 84 I/O lines. There are 24 ADC inputs, and the ADC can convert up to 1 million samples per second.

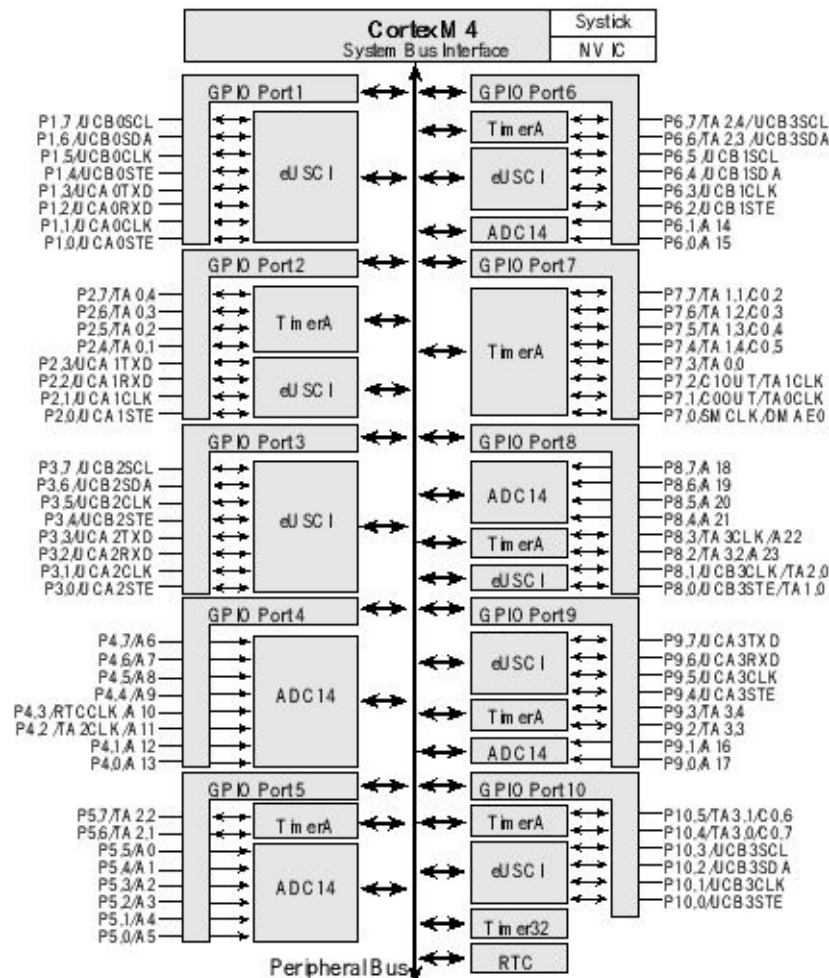


Figure 1.22. I/O port pins for the MSP432P401R microcontroller. (Six pins on Port J not shown).

The MSP432 LaunchPad evaluation board (Figure 1.23) is a low-cost development board available as part number MSP-EXP432P401R from [www.ti.com](http://www.ti.com) and from regular electronic distributors like Digikey, Mouser, element14, and Avnet. The board includes XDS110-ET, an open-source onboard debugger, which allows programming and debugging of the MSP432 microcontroller. The USB interface is used by the debugger and includes a serial channel.

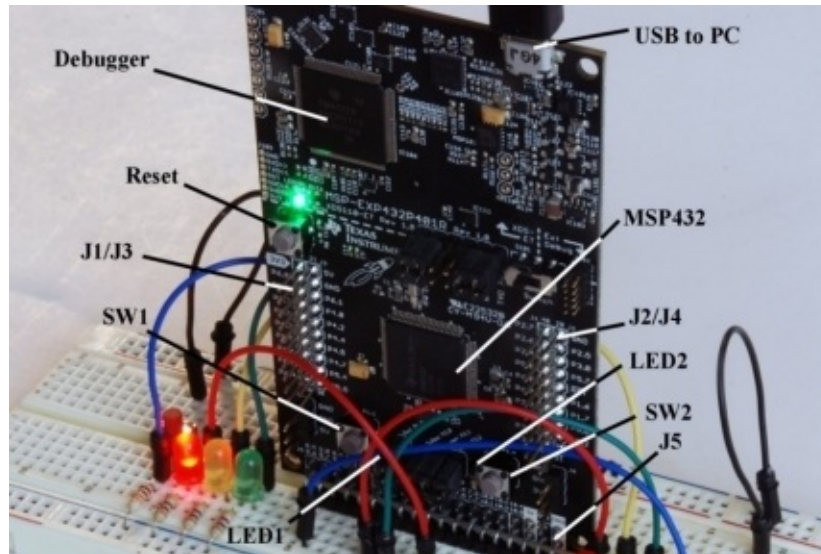


Figure 1.23. LaunchPad based on the MSP432P401RIPZ.

The MSP432 LaunchPad evaluation board has two switches, one 3-color LED and one red LED, as shown in Figure 1.24. The switches are negative logic and will require activation of the internal pull-up resistors. In this class we will not use the switches and LEDs on the LaunchPad, but rather focus on the hardware provided by the MK-II BoosterPack.

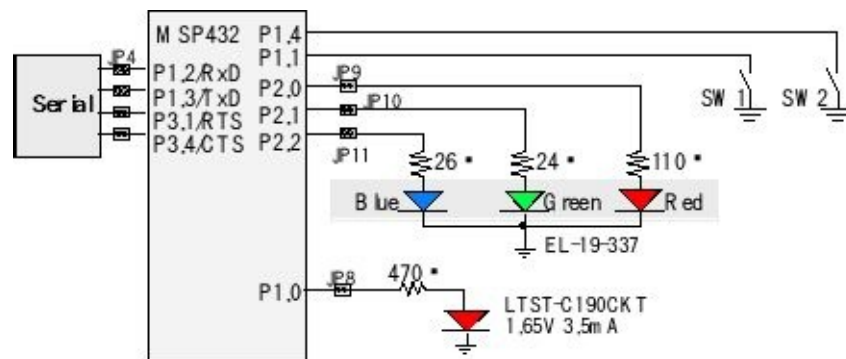


Figure 1.24. Switch and LED interfaces on the LaunchPad Evaluation Board. The jumpers can be removed so the corresponding pin can be used without connection to the external circuits.

The LaunchPad has four 10-pin connectors, labeled as J1 J2 J3 J4 in Figure 1.25, to which you can attach your external signals. The top side of these connectors has male pins, and the bottom side has female sockets.

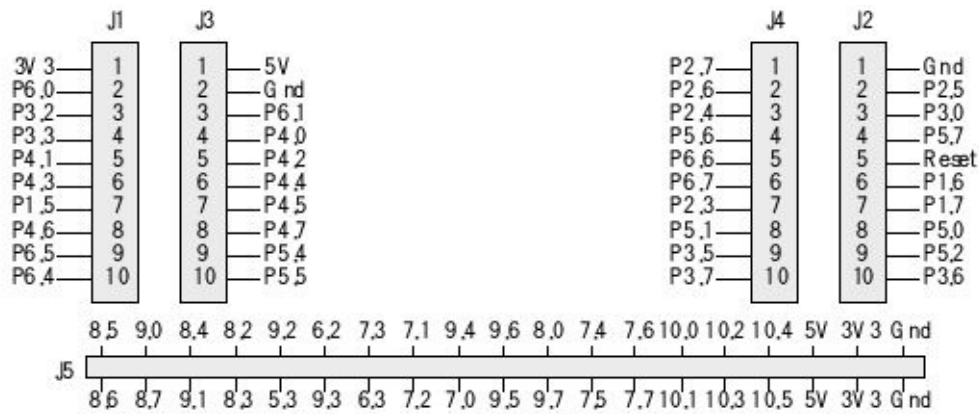


Figure 1.25. Interface connectors on the MSP432 LaunchPad Evaluation Board, 67 I/O pins.

## 1.4.5. Interfacing to a LaunchPad

The LaunchPad ecosystem allows boards to stack together to make a layered system, see Figure 1.26. The engineering community has developed BoosterPacks, which are pre-made external devices that will plug into this 40-pin connector. In addition to the 40-pin header on all LaunchPads, the MSP432 and TM4C1294 LaunchPads have additional headers on the end.

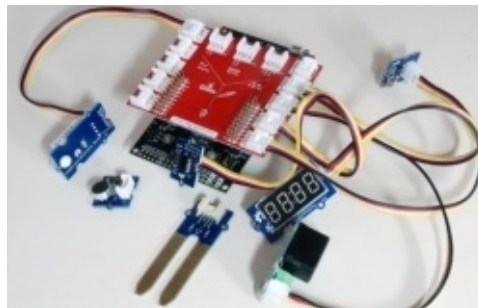
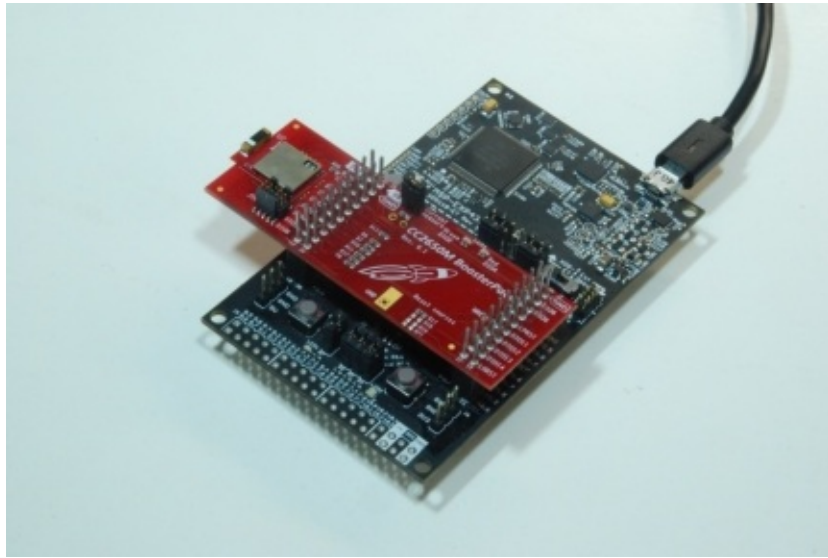


Figure 1.26. An embedded system with MSP432 LaunchPad and a Grove BoosterPack from Seeedstudio.

There are a number of good methods to connect external circuits to the LaunchPad. One method is to purchase a male to female jumper cable (e.g., item number 826 at [www.adafruit.com](http://www.adafruit.com)). A second method is to solder a solid wire into a female socket (e.g., Hirose DF11-2428SCA) creating a male to female jumper wire. The third method is to use BoosterPacks, so you will not need to connect individual wires to the LaunchPad. Figure 1.27 shows the MSP432 with a CC2650 BoosterPack.



*Figure 1.27. A MSP432 LaunchPad with a BOOSTXL-CC2650MA BoosterPack.*

---

## 1.5. ARM Cortex-M Assembly Language

This section focuses on the ARM Cortex-M assembly language. There are many ARM processors, and this book focuses on Cortex-M microcontrollers, which executes Thumb instructions extended with Thumb-2 technology. This section does not present all the Thumb instructions. Rather, we present a few basic instructions. In particular, we will show only twelve instructions, which will be both necessary and sufficient to construct your operating system. For further details, please refer to the appendix or to the ARM Cortex-M Technical Reference Manual.

### 1.5.1. Syntax

Assembly instructions have four fields separated by spaces or tabs as illustrated in Figure 1.28.

*Labels:* The label field is optional and starts in the first column and is used to identify the position in memory of the current instruction. You must choose a unique name for each label.

*Opcodes or pseudo-ops:* The opcode field specifies which processor command to execute. The twelve op codes we will present in this book are **LDR STR MOV PUSH POP B BL BXADD SUB CPSID** and **CPSIE**. If there is a label there must be at least one space or one tab between the label and the opcode. If there is no label then there must be at least one space or one tab at the beginning of the line. There are also pseudo-ops that the assembler uses to control features of the assembly process. Examples of pseudo-ops you will encounter in this class are **AREA EQU IMPORT EXPORT** and **ALIGN**. An op code generates machine instructions that get executed by the processor at run time, while a pseudo-op code generates instructions to the assembler that get interpreted at assembly time.

*Operands:* The operand field specifies where to find the data to execute the instruction. Thumb instructions have 0, 1, 2, 3, or more operands, separated by commas.

*Comments:* The comment field is optional and is ignored by the assembler, but allows you to describe the software, making it easier to understand. You can add optional spaces between operands in the operand field. However, a semicolon must separate the operand and comment fields. Good programmers add comments to explain what you are doing, why you are doing it, how it was tested, and how to change it in the future. Everything after the semicolon is a comment.



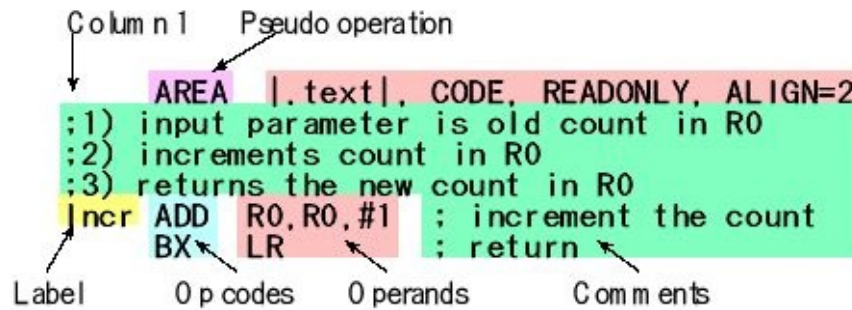


Figure 1.28. Assembly instructions have four fields: labels, opcodes, operands, and comments.

The **assembler** translates assembly source code into **object code**, which are the machine instructions executed by the processor. All object code is halfword-aligned. With Thumb-2, instructions can be 16 or 32 bits wide, and the program counter bit 0 will always be 0. The **listing** is a text file containing a mixture of the object code generated by the assembler together with our original source code.

Address	Object code	Label	Opcode	Operand	comment
0000006A	F100 0001	Incr	ADD	R0,R0,#1	; increment the count
0000006E	4770	BX	LR		; return

When we **build** a project all files are assembled or compiled, then linked together. The address values shown in the listing are the relative to the particular file being assembled. When the entire project is built, the files are linked together, and the **linker** decides exactly where in memory everything will be. After building the project, it can be downloaded, which programs the object code into flash ROM.

In general, the assembler creates for each label an entry in the symbol table that maps the symbolic label to the address in memory of that line of code. The exception to this rule is when a label is used with the **EQU** pseudo-op. The result of an **EQU** pseudo-op is to place an entry in the symbol table mapping the symbolic label with the value of the operand.

## 1.5.2. Addressing modes and operands

A fundamental issue in software design is the differentiation between data and addresses. Another name for address is **pointer**. It is in assembly language programming in general and addressing modes in specific that this differentiation becomes clear. When we put the number 1000 into Register R0, whether this is data or address depends on how the 1000 is used.

The **addressing mode** is the format the instruction uses to specify the memory location to read or write data. We will see five addressing modes in this class:

Immediate	Data within the instruction	<b>MOV R0,#1</b>
Indexed	Data pointed to by register	<b>LDR R0,[R1]</b>
Indexed with offset	Data pointed to by register	<b>LDR R0,[R1,#4]</b>
PC-relative	Location is offset relative to PC	<b>BL Incr</b>
Register-list	List of registers	<b>PUSH {R4,LR}</b>

*No addressing mode:* Some instructions operate completely within the processor and require no memory data fetches. For example, the **ADD R1,R2,R3** instruction performs  $R2+R3$  and stores the sum into R1.

*Immediate addressing mode:* If the data is found in the instruction itself, like **MOV R0,#1**, the instruction uses immediate addressing mode.

*Indexed addressing mode:* A register that contains the address or location of data is called a **pointer** or **index** register. Indexed addressing mode uses a register pointer to access memory. There are many variations of indexed addressing. In this class, you will use two types of indexed addressing. The form **[Rx]** uses Register **Rx** as a pointer, where **Rx** is any of the Registers from R0 to R12. The second type you will need is called indexed with offset, which has the form **[Rx,#n]**, where **n** is a number from -255 to 4095. This addressing mode will access memory at **Rx+n**, without modifying **Rx**.

*PC-relative addressing mode:* The addressing mode that uses the PC as the pointer is called PC-relative addressing mode. It is used for branching, for calling functions, and accessing constant data stored in ROM. The addressing mode is called PC-relative because the machine code contains the address difference between where the program is now and the address to which the program will access.

There are many more addressing modes, but for now, these few addressing modes, as illustrated below, are enough to get us started.

**Checkpoint 1.15:** What does the addressing mode specify?

**Checkpoint 1.16:** How does the processor differentiate between data and addresses?

### 1.5.3. List of twelve instructions

We will only need 12 assembly instructions in order to design our own real-time operating system. The following lists the load and store instructions we will need.

**LDR Rd, [Rn] ; load 32-bit memory at [Rn] to Rd**

**STR Rt, [Rn] ; store Rt to 32-bit memory at [Rn]**

**LDR Rd, [Rn, #n] ; load 32-bit memory at [Rn+n] to Rd**

**STR Rt, [Rn, #n] ; store Rt to 32-bit memory at [Rn+n]**

Let  $M$  be the 32-bit value specified by the 12-bit constant **#imm12**. When **Rd** is absent for add and subtract, the result is placed back in **Rn**. The following lists a few more instructions we will need.

**MOV Rd, Rn ;Rd = Rn**

**MOV Rd, #imm12 ;Rd = M**

**ADD Rd, Rn, Rm ;Rd = Rn + Rm**

**ADD Rd, Rn, #imm12 ;Rd = Rn + M**

**SUB Rd, Rn, Rm ;Rd = Rn - Rm**

**SUB Rd, Rn, #imm12 ;Rd = Rn - M**

**CPSID I ;disable interrupts, I=1**

**CPSIE I ;enable interrupts, I=0**

Normally the computer executes one instruction after another in a linear fashion. In particular, the next instruction to execute is typically found immediately following the current instruction. We use branch instructions to deviate from this straight line path. These branches use PC-relative addressing.

**B label ;branch to label**

**BX Rm ;branch indirect to location specified by Rm**

**BL label ;branch to subroutine at label**

These are the push and pop instructions we will need

**PUSH {Rn,Rm} ; push Rn and Rm onto the stack**

**PUSH {Rn-Rm} ; push all registers from Rn to Rm onto stack**

**POP {Rn,Rm} ; pop two 32-bit numbers off stack into Rn, Rm**

**POP {Rn-Rm} ; pop multiple 32-bit off stack to Rn - Rm**

When pushing and popping multiple registers, it does not matter the order specified in the instruction. Rather, the registers are stored in memory such that the register with the smaller number is stored at the address with a smaller value. For example, consider the execution of **PUSH {R1,R4-R6}**. Assume the registers R1, R4, R5, and R6 initially contain the values 1, 4, 5, and 6 respectively. Figure 1.29 shows the value from lowest-numbered R1 is positioned at the lowest stack address. If four entries are popped with the **POP {R0,R2,R7,R9}** instruction, the value from the lowest stack address is loaded into the lowest-numbered R0.

**Observation:** To push 32-bit data on the stack, first the SP is decremented by 4, and then the data are stored from a register to the RAM location pointed to by SP.

**Observation:** To pop 32-bit data from the stack, first the data are read from the RAM location pointed to by the SP into a register, and then the SP is incremented by 4.

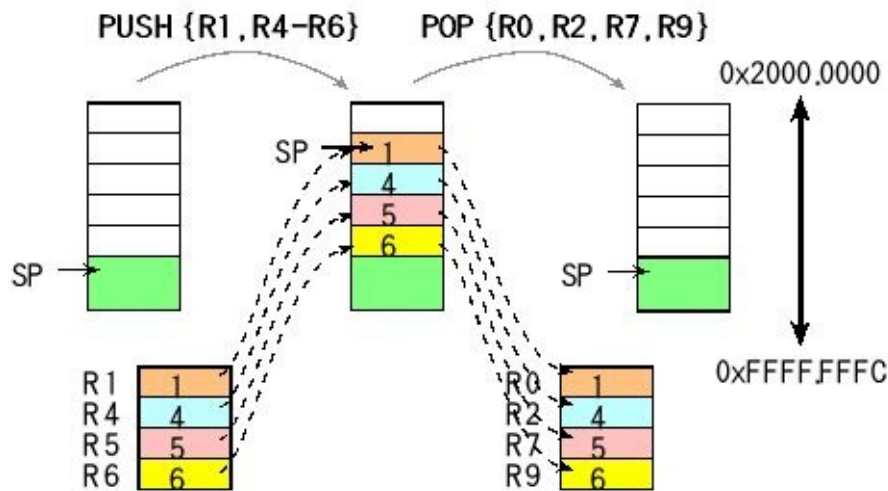


Figure 1.29. Stack drawings showing how multiple registers are pushed and popped.

**Checkpoint 1.17:** How is the SP modified by the **PUSH {R1,R4-R6}** instruction?

## 1.5.4. Accessing memory

One of the basic operations we must perform is reading and writing global variables. Since all calculations are performed in registers, we must first bring the value into a register, modify the register value, and then store the new value back into memory. Consider a simple operation of incrementing a global variable in both C and assembly language. Variables can exist anywhere in RAM, however for this illustration assume the variable **count** is located in memory at 0x20000100. The first **LDR** instruction gets a pointer to the variable in R0 as illustrated in Figure 1.30. This means R0 will have the value 0x20000100. This value is a pointer to the variable **count**. The way it actually works is the assembler places a constant 0x20000100 in code space and translates the **=count** into the correct PC-relative access to the constant (e.g., **LDR R0,[PC,#28]**). The second **LDR** dereferences the pointer to fetch the value of the variable into R1. More specifically, the second **LDR** will read the 32-bit contents at 0x20000100 and put it in R1. The **ADD** instruction increments the value, and the **STR** instruction writes the new value back into the global variable. More specifically, the **STR** instruction will store the 32-bit value from R1 into memory at 0x20000100.

<b>LDR R0,=count ;address of count</b>	
<b>LDR R1,[R0] ;value of count</b>	<b>count = count+1;</b>
<b>ADD R1,R1,#1</b>	
<b>STR R1,[R0] ;store new value</b>	

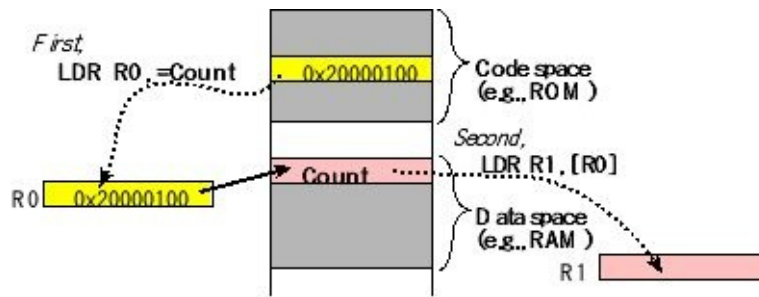


Figure 1.30. Indexed addressing using R0 as a register pointer to access memory. Data is moved into R1. Code space is where we place programs, and data space is where we place variables. The dotted arrows in this figure represent the motion of information, and the solid arrow is a pointer.

Let's work through code similar to what we will use in Chapter 3 as part of our operating system. The above example used indexed addressing with an implicit offset of 0. However, you will also need to understand indexed addressing with an explicit offset. In this example, assume **RunPt** points to a linked list as shown in Figure 1.31. A node of the **list** is a structure (struct in C) with multiple entries of different types. A **linked list** is a set of nodes where one of the entries of the node is a pointer or link to another node of the same type. In this example, the second entry of the list is a pointer to the next node in the list. Figure 1.31 shows three of many nodes that are strung together in a sequence defined by their pointers.

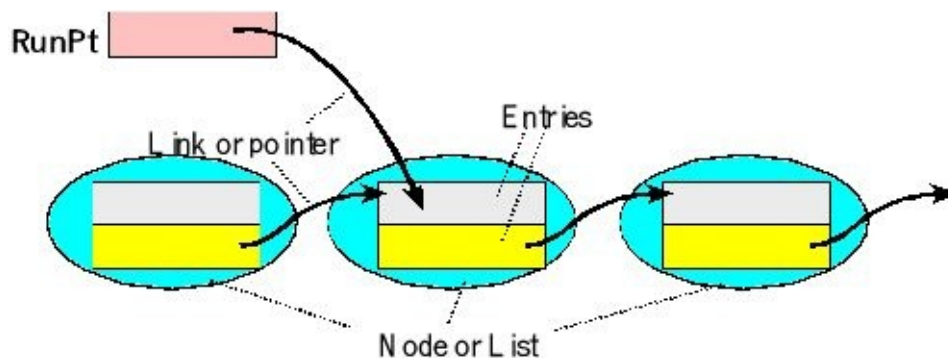


Figure 1.31. A linked list where the second entry is a pointer to the next node. Arrows are pointers or links, and dotted lines are used to label components in the figure.

As our operating system runs it will need to traverse the list. **RunPt** will always point to a node in the list. However, we may wish to change it to point to the next node in the list. In C, we would execute **RunPt=RunPt->next;** However, in assembly this translates to

```

LDR R1,=RunPt ; R1 points to variable RunPt, PC-rel
LDR R0,[R1] ; R0= value of variable RunPt
LDR R2,[R0,#4] ; next entry
STR R2,[R1] ; update RunPt

```

Figure 1.32 draws the action caused by above the four instructions. Assume

initially **RunPt** points to the middle node of the list. Each entry of the node is 32 bits or four bytes of memory. The first two instructions read the value of **RunPt** into R0. Since **RunPt** points to the middle node in the linked list in this figure, R0 will also point to this node. Since each entry is 4 bytes, R0+4 points to the second entry, which is the next pointer. The instruction **LDR R2,[R0,#4]** will read the 32-bit value pointed to by R0+4 and place it in R2. Even though the memory address is calculated as R0+4, the Register R0 itself is not modified by this instruction. R2 now points to the right-most node in the list. The last instruction updates **RunPt** so it now points to the right-most node shown in the Figure 1.32.

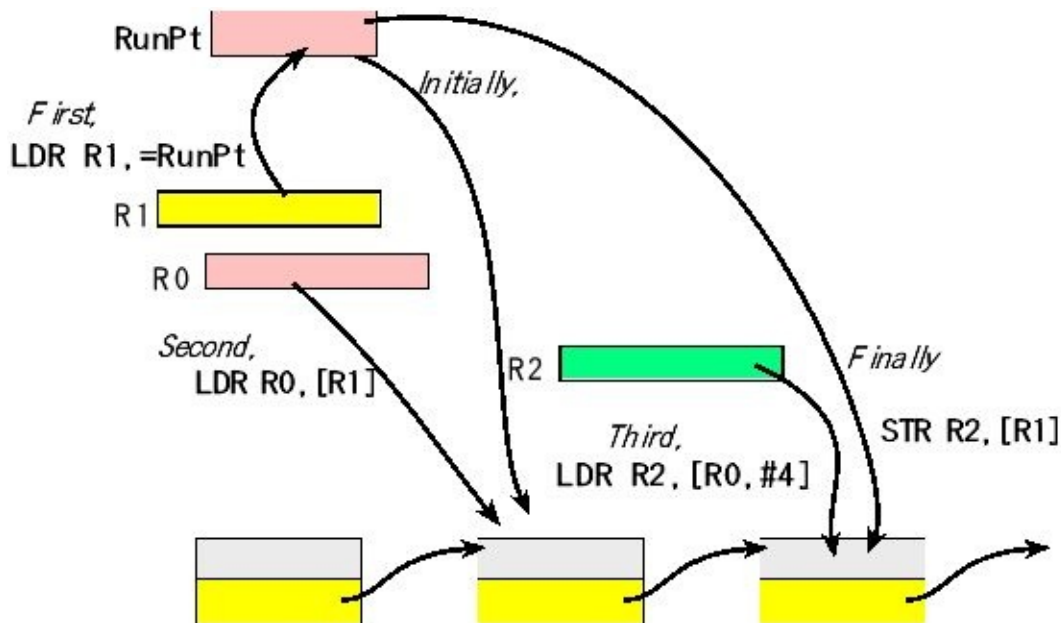


Figure 1.32. An example of indexed addressing mode with offset, data is in memory. Arrows in this figure represent pointers (not the motion of information).

A really important concept. We use the **LDR** instruction to load data from RAM to a register and the **STR** instruction to store data from a register to RAM. In real life, when we *move* a box to the basement, *push* a broom across the floor, *load* bags into the trunk, *store* spoons in a drawer, *pop* a candy into your mouth, or *transfer* employees to a new location, there is a physical object and the action changes the location of that object. Assembly language uses these same verbs, but the action will be different. In most cases, the processor creates a copy of the data and places the copy at the new location. In other words, since the original data still exists in the previous location, there are now two copies of the information. The exception to this memory-access-creates-two-copies-rule is a stack pop. When we pop data from the stack, it no longer exists on the stack leaving us just one copy. Having the information in two places will create a very tricky problem that our operating system must handle.

Let's revisit the simple example of incrementing a global variable. In C, the code would be `count=count+1`; In assembly, the compiler creates code like this:

```

LDR R0,=count ;address of count
LDR R1,[R0] ;value of count
;two copies of count: in memory and in R1
ADD R1,#1
;two copies of count with different values
STR R1,[R0] ;store new value

```

The instruction **LDR R1,[R0]** loads the contents of the variable **count** into R1. At this point, there are two copies of the data, the original in RAM and the copy in R1. After the **ADD** instruction, the two copies have different values. When designing an operating system, we will take special care to handle shared information stored in global RAM, making sure we access the proper copy. In Section 2.2.4, we will discuss in detail the concept of **race conditions** and **critical sections**. These very important problems arise from the problem generated by this concept of having multiple copies of information.

## 1.5.5. Functions

**Subroutines**, **procedures**, and **functions** are programs that can be called to perform specific tasks. They are important conceptual tools because they allow us to develop modular software. The programming languages Pascal, FORTRAN, and Ada distinguish between functions, which return values, and procedures, which do not. On the other hand, the programming languages C, C++, Java, and Lisp do not make this distinction and treat functions and procedures as synonymous. Object-oriented programming languages use the term **method** to describe functions that are part of classes; Objects being instantiation of classes. In assembly language, we use the term subroutine for all subprograms whether or not they return a value. Modular programming allows us to build complex systems using simple components. In this section we present a short introduction on the syntax for defining assembly subroutines. We define a subroutine by giving it a name in the label field, followed by instructions, which when executed, perform the desired effect. The last instruction in a subroutine will be **BX LR**, which we use to return from the subroutine.

The function in Program 1.1 and Figure 1.33 will increment the global variable **count**. The **AREA DATA** directive specifies the following lines are placed in data space (typically RAM). The **SPACE 4** pseudo-op allocates 4 uninitialized bytes. The **AREA CODE** directive specifies the following lines are placed in code space (typically ROM). The **|.text|** connects this program to the C code generated by the compiler. **ALIGN=2** will force the machine code to be halfword-aligned as required.

In assembly language, we will use the **BL** instruction to call this subroutine. At run time, the **BL** instruction will save the return address in the LR register. The return address is the location of the instruction immediately after the **BL** instruction. At the



end of the subroutine, the **BX LR** instruction will get the return address from the LR register, returning the program to the place from which the subroutine was called. More precisely, it returns to the instruction immediately after the instruction that performed the subroutine call. The comments specify the order of execution. The while-loop causes instructions 4–10 to be repeated over and over.

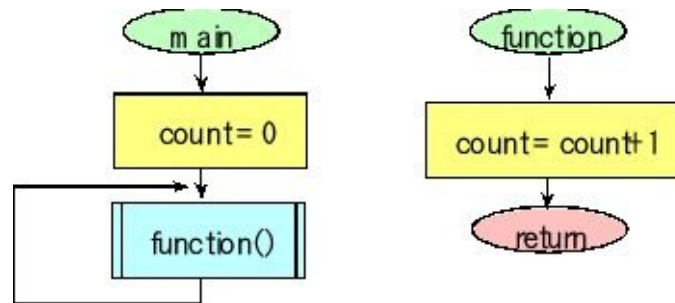


Figure 1.33. A flowchart of a simple function that adds 1 to a global variable.

<pre> <b>AREA DATA</b> count SPACE 4 ; 32-bit data  <b>AREA</b>  .text ,CODE,READONLY,ALIGN=2 <b>function</b>     LDR R0,=count ;5     LDR R1,[R0] ;6 value of count     ADD R1,#1 ;7     STR R1,[R0] ;8 store new value     BX LR ;9 <b>Start</b> LDR R0,=count ;1     MOV R1,#0 ;2     STR R1,[R0] ;3 store new value <b>loop</b> BL function ;4     B loop ;10         </pre>	<pre> <b>uint32_t</b> count  <b>void</b> function(<b>void</b>){     count++; // 5,6,7,8 } // 9  <b>int</b> main(<b>void</b>){     count = 0; // 1,2,3     while(1){         function(); // 4     } // 10 }         </pre>
--	---

Program 1.1. Assembly and C versions that initialize a global array of ten elements. The numbers illustrate the execution sequence.

While using a register (LR) to store the return address is very effective, it does pose a problem if one function were to call a second function. In Program 1.2 **someother** calls **function**. Because the return address is saved in the LR, if one function calls another function it must save the LR before calling and restore the LR after the call. In Program 1.2, the saving and restoring is performed by the **PUSH** and **POP** instructions.

<pre> <b>function</b> ; ..... ; .....     BX LR         </pre>	<pre> <b>void</b> function(<b>void</b>){     // .....     // ..... }         </pre>
--	---



<pre> <b>someother</b> ; .....     <b>PUSH {R4,LR}</b>     <b>BL function</b>     <b>POP {R4,LR}</b> ; .....     <b>BX LR</b> </pre>	<pre> <b>void someother(void){</b>     // .....     <b>function();</b>     // ..... <b>}</b> </pre>
--	---

*Program 1.2. Assembly and C versions that define a simple function.*

**Checkpoint 1.18:** When software calls a function (subroutine), where is the return address saved?

## 1.5.6. ARM Cortex Microcontroller Software Interface Standard

The **ARM Architecture Procedure Call Standard**, AAPCS, part of the **ARM Application Binary Interface (ABI)**, uses registers R0, R1, R2, and R3 to pass input parameters into a C function. R0 is the first parameter, R2 is the second, etc. Functions must preserve the values of registers R4–R11. Also according to AAPCS we place the return parameter in Register R0. AAPCS requires we push and pop an even number of registers to maintain an 8-byte alignment on the stack. In this book, the SP will always be the main stack pointer (MSP), not the Process Stack Pointer (PSP). Recall that all object code is halfword aligned, meaning bit 0 of the PC is always clear. When the **BL** instruction is executed, bits 31–1 of register LR are loaded with the address of the instruction after the **BL**, and bit 0 is set to one. When the **BX LR** instruction is executed, bits 31–1 of register LR are put back into the PC, and bit 0 of LR goes into the T bit. On the ARM Cortex-M processor, the T bit should always be 1, meaning the processor is always in the Thumb state. Normally, the proper value of bit 0 is assigned automatically.

ARM's Cortex Microcontroller Software Interface Standard (CMSIS) is a standardized hardware abstraction layer for the Cortex-M processor series. The purpose of the CMSIS initiative is to standardize a fragmented industry on one superior hardware and software microcontroller architecture.

The CMSIS enables consistent and simple software interfaces to the processor and core MCU peripherals for silicon vendors and middleware providers, simplifying software re-use, reducing the learning curve for new microcontroller developers, and reducing the time to market for new devices. Learn more about CMSIS directly from ARM at [www.onarm.com](http://www.onarm.com).

The CMSIS is defined in close cooperation with various silicon and software vendors and provides a common approach to interface to peripherals, real-time

operating systems, and middleware components. The CMSIS is intended to enable the combination of software components from multiple middleware vendors. The CMSIS components are:

**CMSIS-CORE:** API for the Cortex-M processor core and peripherals. It provides a standardized interface for Cortex-M0, Cortex-M3, Cortex-M4, SC000, and SC300. Included are also SIMD intrinsic functions for Cortex-M4 SIMD instructions.

**CMSIS-DSP:** DSP Library Collection with over 60 Functions for various data types: fixed-point (fractional q7, q15, q31) and single precision floating-point (32-bit). The library is available for Cortex-M0, Cortex-M3, and Cortex-M4. The Cortex-M4 implementation is optimized for the SIMD instruction set.

**CMSIS-RTOS API:** Common API for Real-Time operating systems. It provides a standardized programming interface that is portable to many RTOS and enables software templates, middleware, libraries, and other components that can work across supported RTOS systems.

**CMSIS-SVD:** System View Description for Peripherals. Describes the peripherals of a device in an XML file and can be used to create peripheral awareness in debuggers or header files with peripheral register and interrupt definitions.

**Checkpoint 1.19:** What is the purpose of AAPCS?

## 1.5.7. Conditional execution

If-then-else control structures are commonly found in computer software. If the **BHS** or the **BGE** were to branch, the instruction pipeline would have to be flushed and refilled. In order to optimize execution speed for short if-then and if-then-else control structures, the ARM Cortex-M processor employs conditional execution. The conditional execution begins with the **IT** instruction, which specifies the number of instructions in the control structure (1 to 4) and the conditional for the first instruction. The syntax is

**IT{x{y{z}}}** **cond**

where **x**, **y** and **z** specify the existence of the optional second, third, or fourth conditional instruction respectively. We can specify **x**, **y** and **z** as **T** for execute if true or **E** for else. The **cond** field choices are listed in Table 1.6.

<i>Suffix</i>	<i>Flags</i>	<i>Meaning</i>
<b>EQ</b>	Z = 1	Equal
<b>NE</b>	Z = 0	Not equal
<b>CS</b> or <b>HS</b>	C = 1	Higher or same, unsigned ≥
<b>CC</b> or <b>LO</b>	C = 0	Lower, unsigned <

<b>MI</b>	N = 1	Negative
<b>PL</b>	N = 0	Positive or zero
<b>VS</b>	V = 1	Overflow
<b>VC</b>	V = 0	No overflow
<b>HI</b>	C = 1 and Z = 0	Higher, unsigned >
<b>LS</b>	C = 0 or Z = 1	Lower or same, unsigned ≤
<b>GE</b>	N == V	Greater than or equal, signed ≥
<b>LT</b>	N != V	Less than, signed <
<b>GT</b>	Z = 0 and N = V	Greater than, signed >
<b>LE</b>	Z = 1 or N != V	Less than or equal, signed ≤
<b>AL</b>	Can have any value	Always. This is the default when no suffix is specified.

**Table 1.6. Condition code suffixes used to optionally execution instruction.**

The conditional suffixes for the 1 to 4 following instruction must match the conditional field of the **IT** instruction. In particular, the conditional for the true instructions exactly match the conditional for the **IT** instruction. Furthermore, the else instructions must have the logical complement conditional. If the condition is true the instruction is executed. If the condition is false, the instruction is fetched, but not executed. The following illustrates the use of *if-then* conditional execution. The two T's in **ITT** means there are two true instructions.

```

Change LDR R1,=Num ; R1 = &Num
      LDR R0,[R1] ; R0 = Num
      CMP R0,#25600
      ITT LO
      ADDLO R0,R0,#1 ; if(R0<25600) R0 = Num+1
      STRLO R0,[R1] ; if(R0<25600) Num = Num+1
      BX LR ; return

```

The following illustrates the use of *if-then-else* conditional execution. The one T and one E in **ITE** means there is one *true* and one *false* instruction.

```

Change LDR R1,=Num ; R1 = &Num
      LDR R0,[R1] ; R0 = Num
      CMP R0,#100
      ITE LT
      ADDLT R0,R0,#1 ; if(R0< 100) R0 = Num+1
      MOVGE R0,#-100 ; if(R0>=100) R0 = -100
      STR R0,[R1] ; update Num
      BX LR ; return

```

The following assembly converts one hex digit (0–15) in R0 to ASCII in R1. The one T and one E in **ITE** means there is one true and one else instruction.

```

      CMP R0,#9 ; Convert R0 (0 to 15) into ASCII

```

```

ITE GT ; Next 2 are conditional
ADDGT R1,R0,#55 ; Convert 0xA -> 'A'
ADDLE R1,R0,#48 ; Convert 0x0 -> '0'

```

By themselves, the conditional branch instructions do not require a preceding **IT** instruction. However, a conditional branch can be used as the last instruction of an **IT** block. There are a lot of restrictions on **IT**. For more details, refer to the programming reference manual.

This **macro** creates a new assembly instruction that is faster than **MUL**. This approach can be used to multiply by any constant in the form of  $2^n \pm 1$ . If  $x$  is a variable, then  $15x = (x < 4) \cdot x$ .

```

MACRO
MUL15 $Rd,$Rn
RSB $Rd,$Rn,$Rn,LSL #4
MEND

```

This approach can also be used to multiply by any constant in the form of  $1 \pm 2^{-n}$ . For example, to multiply by  $7/8$  we implement  $x - (x > 3)$ . The **macro** **MUL7\_8** is unsigned multiply by  $7/8$ .

```

MACRO
MUL7_8 $Rd,$Rn
SUB $Rd,$Rn,$Rn,LSR #3
MEND

```

## 1.5.8. Stack usage

The stack can be used to store temporary information. If a subroutine modifies a register, it is a matter of programmer style as to whether or not it should save and restore the register. According to AAPCS a subroutine can freely change R0–R3 and R12, but save and restore any other register it changes. In particular, if one subroutine calls another subroutine, then it must save and restore the LR. AAPCS also requires pushing and popping multiples of 8 bytes, which means an even number of registers. In the following example, assume the function modifies register R0, R4, R7, R8 and calls another function. The programming style dictates registers R4 R7 R8 and LR be saved. Notice the return address is pushed on the stack as LR, but popped off into PC. When multiple registers are pushed or popped, the data exist in memory with the lowest numbered register using the lowest memory address. In other words, the registers in the { } can be specified in any order. Of course remember to balance the stack by having the same number of pops as pushes.

```

Func PUSH {R4,R7,R8,LR} ; save registers as needed
      ; body of the function

```

## POP {R4,R7,R8,PC} ; restore registers and return

The ARM processor has a lot of registers, and we appropriately should use them for temporary information such as function parameters and local variables. However, when there are a lot of parameters or local variables we can place them on the stack. Program 1.3 allocates a 40-byte localbuffer on the stack. The **SUB** instruction allocates 10 words on the stack. Figure 1.34 shows the stack before and after the allocation. The SP points to the first location of **data**. The local variable **i** is held in R0. The flexible second operand for the STR instruction uses SP as the base pointer, and R0\*4 as the offset. The **ADD** instruction deallocates the local variable, balancing the stack.

### // C language implementation

```
void Set(void){
uint32_t data[10];
int i;
  for(i=0; i<10; i++){
    data[i] = i;
  }
}
```

```
Set SUB sp,sp,#0x28 ;allocate
    MOVS r0,#0x00 ;i=0
    B test
loop STR r0,[sp,r0,LSL #2]
    ADDS r0,r0,#1 ;i++
test CMP r0,#0x0A
    BLT loop
    ADD sp,sp,#0x28 ;deallocate
    BX LR
```

Program 1.3. Assembly and C versions that initialize a local array of ten elements.

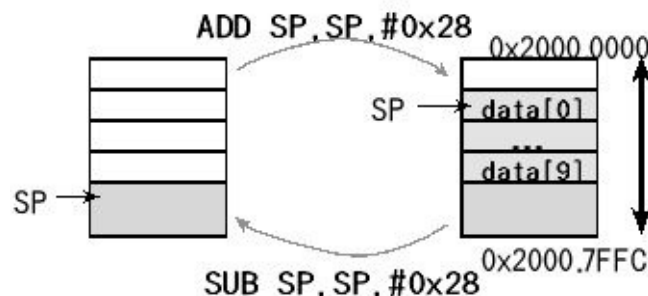


Figure 1.34. A stack picture showing a local array of ten elements (40 bytes).

We will also use the stack to save program state during interrupt processing.

## 1.5.9. Floating-point math

If the range of numbers is unknown or large, then the numbers must be represented in a floating-point format. Conversely, we can use fixed point when the range of values is small and known. The IEEE Standard for Binary Floating-Point Arithmetic or ANSI/IEEE Std 754-1985 is the most widely-used format for floating-point numbers. There are three common IEEE formats: single-precision (32-bit), double-precision (64-bit), and double-extended precision (80-bits). The 32-bit short real format as implemented by the TM4C is presented here. The floating-point format,  $f$ , for the single-precision data type is shown in Figure 1.35. Computers use binary floating point because it is faster to shift than it is to multiply/divide by 10.

Bit 31          Mantissa sign,  $s=0$  for positive,  $s=1$  for negative  
 Bits 30:23      8-bit biased binary exponent  $0 \leq e \leq 255$   
 Bits 22:0       24-bit mantissa,  $m$ , expressed as a binary fraction,  
 A binary 1 as the most significant bit is implied.  
 $m = 1.m_1m_2m_3\dots m_{23}$

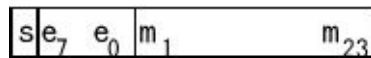


Figure 1.35. 32-bit single-precision floating-point format.

The value of a single-precision floating-point number is

$$f = (-1)^s \cdot 2^{e-127} \cdot m$$

The range of values that can be represented in the single-precision format is about  $\pm 10^{-38}$  to  $\pm 10^{+38}$ . The 24-bit mantissa yields a precision of about 7 decimal digits. The floating-point value is zero if both  $e$  and  $m$  are zero. Because of the sign bit, there are two zeros, positive and negative, which behave the same during calculations.

There are some special cases for floating-point numbers. When  $e$  is 255, the number is considered as plus or minus infinity, which probably resulted from an overflow during calculation. When  $e$  is 0, the number is considered as **denormalized**. The value of the mantissa of a denormalized number is less than 1. A denormalized short result number has the value,

$$f = (-1)^s \cdot 2^{-126} \cdot m \quad \text{where } m = 0.m_1m_2m_3\dots m_{23}$$

**Observation:** The floating-point zero is stored in denormalized format.

When two floating-point numbers are added or subtracted, the smaller one is first **unnormalized**. The mantissa of the smaller number is shifted right and its exponent is incremented until the two numbers have the same exponent. Then, the mantissas are

added or subtracted. Lastly, the result is normalized. To illustrate the floating-point addition, consider the case of  $10+0.1$ . First, we show the original numbers in floating-point format. The mantissa is shown in binary format.

$$\begin{aligned} 10.0 &= (-1)^0 \cdot 2^3 \cdot 1.010000000000000000000000 \\ + 0.1 &= (-1)^0 \cdot 2^{-4} \cdot 1.10011001100110011001101 \end{aligned}$$

Every time the exponent is incremented the mantissa is shifted to the right. Notice that 7 binary digits are lost. The 0.1 number is unnormalized, but now the two numbers have the same exponent. Often the result of the addition or subtraction will need to be normalized. In this case the sum did not need normalization.

$$\begin{aligned} 10.0 &= (-1)^0 \cdot 2^3 \cdot 1.010000000000000000000000 \\ + 0.1 &= (-1)^0 \cdot 2^3 \cdot 0.0000001100110011001100110011001101 \\ 10.1 &= (-1)^0 \cdot 2^3 \cdot 1.01000011001100110011001 \end{aligned}$$

When two floating-point numbers are multiplied, their mantissas are multiplied and their exponents are added. When dividing two floating-point numbers, their mantissas are divided and their exponents are subtracted. After multiplication and division, the result is normalized.

**Roundoff** is the error that occurs as a result of an arithmetic operation. For example, the multiplication of two 64-bit mantissas yields a 128-bit product. The final result is normalized into a normalized floating-point number with a 64-bit mantissa. Roundoff is the error caused by discarding the least significant bits of the product. Roundoff during addition and subtraction can occur in two places. First, an error can result when the smaller number is shifted right. Second, when two  $n$ -bit numbers are added the result is  $n+1$  bits, so an error can occur as the  $n+1$  sum is squeezed back into an  $n$ -bit result.

**Truncation** is the error that occurs when a number is converted from one format to another. For example, when an 80-bit floating-point number is converted to 32-bit floating-point format, 40 bits are lost as the 64-bit mantissa is truncated to fit into the 24-bit mantissa. Recall, the number 0.1 could not be exactly represented as a short real floating-point number. This is an example of truncation as the true fraction was truncated to fit into the finite number of bits available.

If the range is known and small and a fixed-point system can be used, then a 32-bit fixed-point number system will have better resolution than a 32-bit floating-point system. For a fixed range of values (i.e., one with a constant exponent), a 32-bit floating-point system has only 23 bits of precision, while a 32-bit fixed-point system has 9 more bits of precision.

Figure 1.36 shows the floating-point registers on the Cortex M4. Software can access these registers in any combination of 32 single-precision registers named S0 to S31 or 16 double-precision registers D0 to D15. In particular, registers S0 and S1 are the same as register D0. This section will focus on single precision floating-point operations.

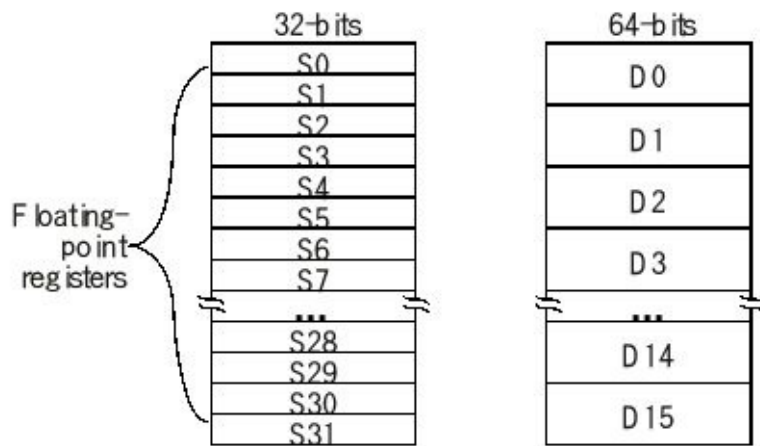


Figure 1.36. The TM4C has 32 single-precision floating-point registers that overlap with 16 double-precision floating-point registers.

The following lists the general form for some of the load and store instructions. Because the constant is stored into memory, and the assembly creates a PC relative access, the constant can be any single-precision floating-point value. **St Sd Sn** and **Sm** represent any of the 32 single-precision floating-point registers. **Rn** and **Rd** are regular integer registers.

**VLDR.F32 Sd, [Rn] ; load 32-bit float at [Rn] to Sd**

**VSTR.F32 St, [Rn] ; store 32-bit St to memory at [Rn]**

**VLDR.F32 Sd, [Rn, #n] ; load 32-bit memory at [Rn+n] to Sd**

**VSTR.F32 St, [Rn, #n] ; store 32-bit St to memory [Rn+n]**

**VLDR.F32 Sd, =constant ; load 32-bit constant into Sd**

The move instructions get their data from the machine instruction or from within the processor and do not require additional memory access instructions. The immediate value is any number that can be expressed as  $\pm n \cdot 2^{-r}$ , where  $16 \leq n \leq 31$ , and  $0 \leq r \leq 7$ .

**VMOV.F32 Sd, Sn ; set Sd equal to the value in Sn**

**VMOV.F32 Sd, #imm ; set Sd equal to imm**

**VMOV Rd, Sn ; set Rd equal to the value in Sn**

**VMOV Sd, Rn ; set Sd equal to the value in Rn**

These are some of the arithmetic operations, which operate on the floating-point registers. Arithmetic operations can cause overflow, underflow, divide by zero floating-point exceptions. In particular, bits in the **SYSEXC\_RIS\_R** register will get set if there is a floating-point error.

**VADD.F32 Sd, Sn, Sm ; set Sd equal to Sn+Sm**

**VSUB.F32 Sd, Sn, Sm ; set Sd equal to Sn-Sm**

**VMUL.F32 Sd, Sn, Sm ; set Sd equal to Sn\*Sm**



**VDIV.F32 Sd, Sn, Sm ; set Sd equal to Sn/Sm**

**VNEG.F32 Sd, Sm ; set Sd equal to -Sm**

**VABS.F32 Sd, Sm ; set Sd equal to the absolute value of Sm**

**VSQRT.F32 Sd, Sm ; set Sd equal to the square root of Sm**

The following example implements a digital 60 Hz notch filter (see Section 6.4). The new ADC input is passed by value in register S0 and the filter output is returned by value also in register S0. In C, we define a single-precision floating-point variable using **float**.

```
float y,y1,y2; // outputs
float x,x1,x2; // input
// fs = 1000 Hz
// cutoff 60 Hz
// alpha = 0.99
float Notch60Hz(float in){
    x2 = x1; x1 = x; x = in;
    y2 = y1; y1 = y;
    y = x
        - 1.8595529717765*x1
        + x2
        + 1.84095744205874*y1
        - 0.9801*y2;
    return y;
}
AREA DATA, ALIGN=2
y SPACE 4 ; current filter output
y1 SPACE 4 ; filter output 1ms ago
y2 SPACE 4 ; filter output 2ms ago
x SPACE 4 ; current filter input
x1 SPACE 4 ; input 1ms ago
x2 SPACE 4 ; input 2ms ago
AREA |.text|,CODE,READONLY,ALIGN=2
THUMB
; Input: S0 is new input
; Output: S0 is filter output
Notch60Hz
    LDR    R0,=x
    VLDR.F32 S1,[R0,#4] ;read previous x1
    VSTR.F32 S1,[R0,#8] ;S1 is x2
    VLDR.F32 S2,[R0,#0] ;read previous x
    VSTR.F32 S2,[R0,#4] ;S2 is x1
    VSTR.F32 S0,[R0,#0] ;S0 is x = in
    LDR    R1,=y
    VLDR.F32 S3,[R1,#4] ;read previous y1
    VSTR.F32 S3,[R1,#8] ;S3 is y2
    VLDR.F32 S4,[R1,#0] ;read previous y
    VSTR.F32 S4,[R1,#4] ;S4 is y1
    VLDR.F32 S5,=-1.8595529717765
    VMUL.F32 S2,S2,S5
    VADD.F32 S0,S0,S2 ;-1.8595529717765*x1
    VADD.F32 S0,S0,S1 ;+x2
    VLDR.F32 S5,=1.84095744205874
    VMUL.F32 S4,S4,S5
    VADD.F32 S0,S0,S4;+1.84095744205874*y1
    VLDR.F32 S5,=-0.9801
    VMUL.F32 S3,S3,S5
```

```
VADD.F32 S0,S0,S3 ; -0.9801*y2
VSTR.F32 S0,[R1,#0] ;set y
BX LR
```

*Program 1.4. Floating-point function to a 60 Hz IIR digital filter (assembly program executes in 43 cycles).*

**Observation:** If you are implementing digital signal processing using floating point math, we strongly recommend implement the functions in assembly so you can specify exactly how the floating point hardware is to be used.

## 1.5.10. Keil assembler directives

We use assembler directives to assist and control the assembly process. The following directives change the way the code is assembled.

<b>AREA CODE</b>	<b>;places code in code space (flash ROM)</b>
<b>AREA DATA</b>	<b>;places objects in data space (RAM)</b>
<b>THUMB</b>	<b>;uses Thumb instructions</b>
<b>ALIGN</b>	<b>;skips 0 to 3 bytes to make next word aligned</b>
<b>END</b>	<b>;end of file</b>

The following directives can add variables and constants.

<b>DCB expr{,expr}</b>	<b>;places 8-bit byte(s) into memory</b>
<b>DCW expr{,expr}</b>	<b>;places 16-bit halfword(s) into memory</b>
<b>DCD expr{,expr}</b>	<b>;places 32-bit word(s) into memory</b>
<b>SPACE size</b>	<b>;reserves size bytes, uninitialized</b>

The **EQU** directive gives a symbolic name to a numeric constant, a register-relative value or a program-relative value. \* is a synonym for **EQU** . We will use it to define I/O port addresses. For example, these four definitions will be used to initialize and operate Port D.

```
GPIO_PORTD_DATA_R equ 0x400073FC
GPIO_PORTD_DIR_R equ 0x40007400
GPIO_PORTD_DEN_R equ 0x4000751C
SYSCCTL_RCGCGPIO_R equ 0x400FE608
```

In order for another file to access a variable or function in this assembly file we use the **EXPORT** directive. In order for this assembly file to access a variable or function in another file we use the **IMPORT** directive. All C public functions and global variables (no static) are available to be imported into assembly. To import a function into a C file, we define a prototype. To import a global variable into a C file, we define it with an **extern** .

```
uint32_t v2; // global
extern uint32_t v1;
```

```
uint32_t f1(uint32_t in);
```

```
void f2(void){  
    v1 = f1(v1);  
}
```

AREA

```
DATA, ALIGN=2  
    EXPORT v1  
    EXPORT f1  
    IMPORT v2
```

v1 SPACE

```
4 ; global  
    AREA |.text|,CODE,READONLY,ALIGN=2
```

THUMB

```
f1 LDR R1,=v2  
    LDR R2,[R1] ; contents  
    ADD R0,R0,R2  
    BX LR
```

# 1.6. Pointers in C

## 1.6.1. Pointers

At the assembly level, we implement **pointers** using indexed addressing mode. For example, a register contains an address, and the instruction reads or writes memory specified by that address. Basically, we place the address into a register, then use indexed addressing mode to access the data. In this case, the register holds the pointer. Figure 1.37 illustrates three examples that utilize pointers. In this figure, **Pt**, **SP**, **GetPt**, **PutPt** are pointers, where the arrows show to where they point, and the shaded boxes represent data. An **array** or **string** is a simple structure containing multiple equal-sized elements. We set a pointer to the address of the first element, then use indexed addressing mode to access the elements inside. We have introduced the stack previously, and it is an important component of an operating system. The stack pointer (**SP**) points to the top element on the stack. A linked list contains some elements that are pointers themselves. The pointers are used to traverse the data structure. Linked lists will be used through this book to maintain the states of threads in our RTOS. The first in first out (**FIFO**) queue is an important data structure for I/O programming because it allows us to pass data from one module to another. One module puts data into the **FIFO** and another module gets data out of the **FIFO**. There is a **GetPt** that points to the oldest data (to be removed next) and a **PutPt** that points to an empty space (location to be stored into next). The **FIFO** queue will be used excessively in this book.

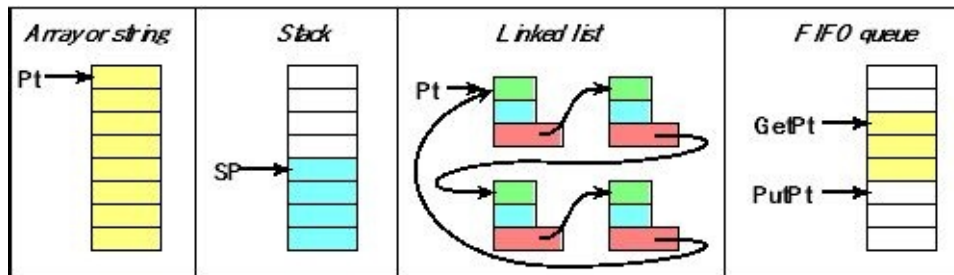


Figure 1.37. Examples of data structures that utilize pointers.

We will illustrate the use of pointers with some simple examples. Consider that we have a global variable called **Count**. This creates a 32-bit space in memory to contain the value of this variable. The **int** declaration means “is a signed 32-bit integer”.

**int Count;**

There are three phases to using pointers: creation, initialization, usage. To create a pointer, we define a variable placing the **\*** before its name. As a convention, we will use “p”, “pt”, or “ptr” in the variable name to signify it is a pointer variable. The **\***

means “is a pointer to”. Therefore, **int \*** means “is a pointer to a signed 32-bit integer”.

**int \*cPt;**

To initialize a pointer, we must set it to point to something. Whenever we make an assignment in C, the type of the value must match the type of the variable. The following executable code makes **cPt** point to **Count** . We see the type of **Count** is signed 32-bit integer, so the type of **&Count** is a pointer to a signed 32-bit integer.

**cPt = &Count;**

Assume we have another variable called **x** , and assume the value of **Count** is 42. Using the pointer is called dereferencing. If we place a **\*cPt** inside an expression, then **\*cPt** is replaced with the value at that address. So this operation will set **x** equal to 42.

**x = (\*cPt);**

If we place a **\*cPt** as the assignment, then the value of the expression is stored into the memory at the address of the pointer. So, this operation will set **Count** equal to 5;

**(\*cPt) = 5;**

We can use the dereferencing operator in both the expression and as the assignment. These operations will increment **Count** .

**(\*cPt) = (\*cPt) + 1;**

**(\*cPt) += 1;**

**(\*cPt)++;**

Functions that require data to be passed by the value they hold are said to use **call-by-value** parameter passing. With an input parameter using call by value, the data itself is passed into the function. For an output parameter using return by value, the result of the function is a value, and the value itself is returned. According to AAPCS, the first four input parameters are passed in R0 to R3 and the output parameter is returned in R0. Alternatively, if you pass a pointer to the data, rather than the data itself, we will be able to pass large amounts of data. Passing a pointer to data is classified as **call-by-reference**. For large amounts of data, call by reference is faster, because the data need not be copied from calling program to the called subroutine. In call by reference, the one copy of the data exists in the calling program, and a pointer to it is passed to the subroutine. In this way, the subroutine actually performs read/write access to the original data. Call by reference is also a convenient mechanism to return data as well. Passing a pointer to an object allows this object (a primitive data type like char, int, or a collection like an array, or a composite struct data type) to be an input parameter and an output parameter.

Our real-time operating system will make heavy use of pointers. In this example, the function is allowed to read and write the original data:

```

void Increment(int *cpt){
    (*cpt) = (*cpt)+1;
}

```

We will also use pointers for arrays, linked-lists, stacks, and first-in-first-out queues. If your facility with pointers is weak, we suggest you review pointers.

**Checkpoint 1.20:** What are pointers and why are they important?

## 1.6.2. Arrays

Figure 1.38 shows an array of the first ten prime numbers stored as 32-bit integers, we could allocate the structure in ROM using

```
int const Primes[10]={1,2,3,5,7,11,13,17,19,23};
```

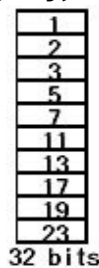


Figure 1.38. Array of ten 32-bit values.

By convention, we define **Primes[0]** as the first element, **Primes[1]** as the second element, etc. The address of the first element can be written as **&Primes[0]** or just **Prime**. In C, if we want the 5<sup>th</sup> element, we use the expression **Primes[4]** to fetch the 7 out of the structure. In C the following two expressions are equivalent, both of which will fetch the contents from the 5<sup>th</sup> element.

```

Primes[4]
*(Primes+4)

```

In C, we define a pointer to a signed 32-bit constant as

```
int const *Cpt;
```

In this case, the **const** does not indicate the pointer is fixed. Rather, the pointer refers to constant 16-bit data in ROM. We initialize the pointer at run time using

```
Cpt = Primes; // Cpt points to Primes
```

or

```
Cpt = &Primes[0]; // Cpt points to Primes
```

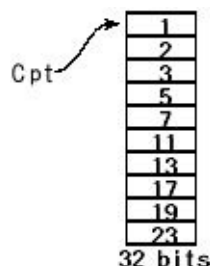


Figure 1.39. *Cpt* is a pointer to an array of ten 32-bit values.

When traversing an array, we often wish to increment the pointer to the next element. To move the pointer to the next element, we use the expression **Cpt++**. In C, **Cpt++**, which is the same thing as **Cpt = Cpt+1**; actually adds four to the pointer because it points to 32-bit words. If the array contained 8-bit data, incrementing the pointer would add 1. If the array contained 16-bit data, incrementing the pointer adds 2. The pointers themselves are always 32-bits on the ARM, but the data could be 1, 2, 4, 8 ... bytes.

As an example, consider the situation where we wish to pass a large amount of data into the function **BubbleSort**. In this case, we have one or more buffers, defined in RAM, which initially contains data in an unsorted fashion. The buffers shown here are uninitialized, but assume previously executed software has filled these buffers with corresponding voltage and pressure data. In C, we could have

```
uint8_t VBuffer[100]; // voltage data
uint8_t PBuffer[200]; // pressure data
```

Since the size of these buffers is more than will fit in the registers, we will use call by reference. In C, to declare a parameter call by reference we use the **\***.

```
void BubbleSort(uint8_t *pt, uint32_t size){
uint32_t i,j; uint8_t data,*p1,*p2;
for(i=1; i<size; i++){
    p1 = pt; // pointer to beginning
    for(j=0; j<size-i; j++){
        p2 = p1+1; // p2 points to the element after p1
        if((*p1) > (*p2)){
            data = (*p1); // swap
            (*p1) = (*p2);
            (*p2) = data;
        }
        p1++;
    }
}
}
```

To invoke a function using call by reference we pass a pointer to the object. These two calling sequences are identical, because in C the array name is equivalent to a pointer to its first element (**VBuffer** is equivalent to **&VBuffer[0]**). Recall that the **&** operator is used to get the address of a variable.

```
void main(void){          void main(void){
    BubbleSort(Vbuffer,100);      BubbleSort(&VBuffer[0],100);
    BubbleSort(Pbuffer,200);      BubbleSort(&PBuffer[0],200);
}                               }
```

One advantage of call by reference in this example is the same buffer can be used

also as the return parameter. In particular, this sort routine re-arranges the data in the same original buffer. Since RAM is a scarce commodity on most microcontrollers, not having to allocate two buffers will reduce RAM requirements for the system.

From a security perspective, call by reference is more vulnerable than call by value. If we have important information, then a level of trust is required to pass a pointer to the original data to a subroutine. Since call by value creates a copy of the data at the time of the call, it is slower but more secure. With call by value, the original data is protected from subroutines that are called.

**Checkpoint 1.21:** If an array has 10 elements, what is the range of index values used to access the data?

### 1.6.3. Linked lists

The linked list is an important data structure used in operating systems. Each element (node) contains data and a pointer to another element as shown in Figure 1.40. Given that a node in the list is a composite of data and a pointer, we use **struct** to declare a composite data type. A composite data type can be made up of primitive data type, pointers and also other composite data-types.

```
struct Node{
    struct Node *Next;
    int Data;
};
typedef struct Node NodeType;
```

In this simple example, the Data field is just a 32-bit number, we will expand our node to contain multiple data fields each storing a specific attribute of the node. There is a pointer to the first element, called the head pointer. The last element in the list has a null pointer in its next field to indicate the end of the list.

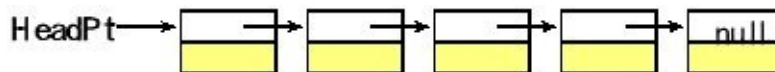


Figure 1.40. A linked list with 5 nodes.

We can create lists statically or dynamically. A statically created list is created at compile time and does not change during the execution of the program.

```
NodeType theList[8] = {
    {&theList[1], 1},
    {&theList[2], 10},
    {&theList[3], 100},
    {&theList[4], 1000},
    {&theList[5], 10000},
    {&theList[6], 100000},
    {&theList[7], 1000000},
    {&theList[8], 10000000}
```



```

    {&theList[7], 1000000},
    {0, 10000000}};
NodeType *HeadPt = theList; // points to first element

```

The following function searches the list to see if a data value exists in the list.

```

int Search(int x){ NodeType *pt;
    pt = HeadPt; // start at beginning
    while(pt){
        if(pt->Data == x) return 1; // found
        pt = pt->Next;
    }
    return 0; // not found
}

```

This example created the linked-list statically. The compiler will generate code prior to running main (called premain) that will initialize the eight nodes. To do this initialization, there will be two copies of the structure: the initial copy in ROM used during premain, and the RAM copy used by the program during execution. If the program needs to change this structure during execution then having two copies is fine. However, if the program does not change the structure, then you could put a single copy in ROM by adding **const** to the definition. In this case, **HeadPt** will be in RAM but the linked list will be in ROM.

```

const struct Node{
    const struct Node *Next;
    int Data;
};
typedef const struct Node NodeType;
NodeType theList[8] = {
    {&theList[1], 1},
    {&theList[2], 10},
    {&theList[3], 100},
    {&theList[4], 1000},
    {&theList[5], 10000},
    {&theList[6], 100000},
    {&theList[7], 1000000},
    {0, 10000000}};
NodeType *HeadPt = theList; // points to first element

```

It is possible to create a linked list dynamically and grow/shrink the list as a program executes. However, in keeping with our goal to design a simple RTOS, we will refrain from doing any dynamic allocation, which would require the management of a heap. Most real-time systems do not allow the heap (malloc and free) to be accessed by the application programmer, because the use of the heap could lead to

**nondeterministic** behavior (the activity of one program affects the behavior of another completely unrelated program).

**Checkpoint 1.22:** What is a linked list and in what ways is it better than an array? In what ways are arrays better?

---

## 1.7. Memory Management

### 1.7.1. Use of the heap

In the previous two volumes, we have seen two types of allocation: permanent allocation in global variables and temporary allocation in local variables. When we allocate local variables in registers or on the stack these variables must be private to the function and cannot be shared with other functions. Furthermore, each time the function is invoked new local variables are created, and data from previous instantiations are not available. This behavior is usually exactly what we want to happen with local variables. However, we can use the **heap** (or **memory manager**) to have temporary allocation in a way that is much more flexible. In particular, we will be able to explicitly define when data are allocated and when they are deallocated with the only restriction being we first allocate, next we use, and then we deallocate. Furthermore, we can control the scope of the data in a flexible manner.

The use of the heap involves two system functions: **malloc** and **free** . When we wish to allocate space, we call **malloc** and specify how many bytes we need. **malloc** will return a pointer to the new object, which we must store in a pointer variable. If the heap has no more space, **malloc** will return a 0, which means null pointer. The heap implements temporary allocation, so when we are done with the data, we return it to the heap by calling **free** . Consider the following simple example with three functions.

```
int32_t *Pt;
void Begin(void){
    Pt = (*int32_t)malloc(4*20); // allocate 20 words
}
void Use(void){ int32_t i;
    for(i = 0; i < 20; i++)
        Pt[i] = i; // put data into array
}
void End(void){
    free(Pt);
}
```

The pointer **Pt** is permanently allocated. The left side of Figure 1.41 shows that initially, even though the pointer exists, it does not point to anything. More specifically, the compiler will initialize it to 0; this 0 is defined as a **nullpointer**, meaning it is not valid. When **malloc** is called the pointer is now valid and points to a 20-word array. The array is inside the heap and **Pt** points to it. Any time after **malloc** is called and before **free** is called, the array exists and can be accessed

via the pointer **Pt** . After you call **free** , the pointer has the same value as before. However, the array itself does not exist. I.e., these 80 bytes do not belong to your program anymore. In particular, after you call **free** , the heap is allowed to allocate these bytes to some other program. Weird and crazy errors will occur if you attempt to dereference the pointer before the array is allocated, or after it is released.

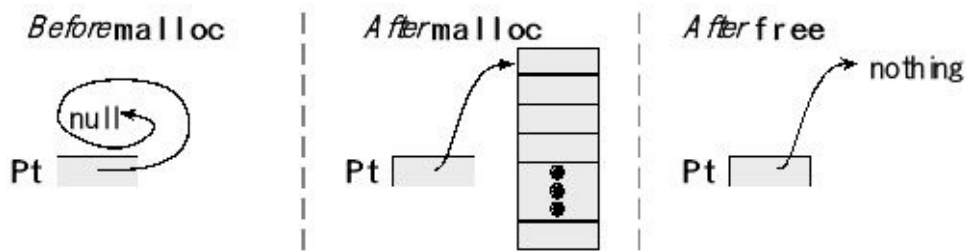


Figure 1.41. The heap is used to dynamically allocate memory.

This array exists and the pointer is valid from when you call **malloc** up until the time you call **free** . In C, the heap does not manage the pointers to allocated block; your program must. If you call **malloc** ten times in a row, you must keep track of the ten pointers you received. The scope of this array is determined by the scope of the pointer, **Pt** . If **Pt** is public, then the array is public. If **static** were to be added to the definition of **Pt** , then the scope of the array is restricted to software within this file. In the following example, the scope of the array is restricted to the one function. Within one execution of the function, the array is allocated, used, and then deallocated, just like a local variable.

```
void Function(void){ int32_t i;
int32_t *pt;
pt = (*int32_t)malloc(4*20); // allocate 20 words
for(i = 0; i < 20; i++)
    pt[i] = i; // put data into array
free(pt);
}
```

A **memory leak** occurs if software uses the heap to allocate space but forgets to deallocate the space when it is finished. The following is an example of a memory leak. Each time the function is called, a block of memory is allocated. The pointer to the block is stored in a local variable. When the function returns, the pointer **pt** no longer exists. This means the allocated block in the heap exists, but the program has no pointer to it. In other words, each time this function returns 80 bytes from the heap are permanently lost.

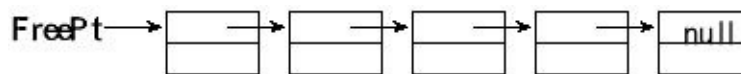
```
void LeakyFunction(void){ int32_t i;
int32_t *pt;
pt = (*int32_t)malloc(4*20); // allocate 20 words
for(i = 0; i < 20; i++)
    pt[i] = i; // put data into array
}
```

**Internal fragmentation** is storage that is allocated for the convenient of the operating system but contains no information. This space is wasted. Often this space is wasted in order to improve speed or provide for a simpler implementation. The fragmentation is called "internal" because the wasted storage is inside the allocated region. **External fragmentation** exists when the largest memory block that can be allocated is less than the total amount of free space in the heap. External fragmentation occurs in simple memory managers because memory is allocated in contiguous blocks. External fragmentation occurs over time as free storage becomes divided into many small pieces. It is a particular problem when an application allocates and deallocates blocks of storage of varying sizes. The result is that although free storage is available, it is effectively unusable because it is divided into pieces that are too small to satisfy the demands of the application. The term "external" refers to the fact that the unusable storage is outside the allocated regions.

**Checkpoint 1.23:** Depending on the microcontroller architecture, it may be faster to access variables allocated on either a 16-bit word or 32-bit boundary. If the compiler skips memory cells in order to align variables, is this internal or external fragmentation?

## 1.7.2. Simple fixed-size heap

In general, the heap manager allows the program to allocate a variable block size, but in this section we will develop a simplified heap manager handles just fixed size blocks. In this example, the block size is specified by the constant **SIZE**. The initialization will create a linked list of all the free blocks (Figure 1.42).



*Figure 1.42. The initial state of the heap has all of the free blocks linked in a list.*

Program 1.5 shows the global structures for the heap. These entries are defined in RAM. **SIZE** is the number of 8-bit bytes in each block. All blocks allocated and released with this memory manager will be of this fixed size. **NUM** is the number of blocks to be managed. **FreePt** points to the first free block.

```
#define SIZE 80
#define NUM 5
#define NULL 0 // empty pointer
int8_t *FreePt;
int8_t Heap[SIZE*NUM];
```

*Program 1.5a. Private global structures for the fixed-block memory manager.*

Initialization must be performed before the heap can be used. Program 1.5b shows the software that partitions the heap into blocks and links them together. **FreePt** points to a linear linked list of free blocks.

```
void Heap_Init(void){
    int8_t *pt;
    FreePt = &Heap[0];
    for(pt=&Heap[0]; pt!=&Heap[SIZE*(NUM-1)]; pt=pt+SIZE){
        *(int32_t *)pt =(int32_t)(pt+SIZE);
    }
    *(int32_t*)pt = NULL;
}
```

*Program 1.5b. Functions to initialize the heap.*

Initially these free blocks are contiguous and in order, but as the manager is used the positions and order of the free blocks can vary. It will be the pointers that will thread the free blocks together. To allocate a block to manager just removes one block from the free list. Program 1.5c shows the allocate and release functions. The **Heap\_Allocate** function will fail and return a null pointer when the heap becomes empty. The **Heap\_Release** returns a block to the free list. This system does not check to verify a released block actually was previously allocated.

```
void *Heap_Allocate(void){int8_t *pt;
    pt = FreePt;
    if (pt != NULL){
        FreePt = (int8_t*) *(int8_t**)pt;
    }
    return(pt);
}
void Heap_Release(void *pt){int8_t *oldFreePt;
    oldFreePt = FreePt;
    FreePt = (int8_t*)pt;
    *(int32_t *)pt = (int32_t)oldFreePt;
}
```

*Program 1.5c. Functions to allocate and release memory blocks.*

**Checkpoint 1.24:** There are 5 blocks in this simple heap. How could the memory manager determine if block I (where  $0 \leq I \leq 4$ ) is allocated or free?

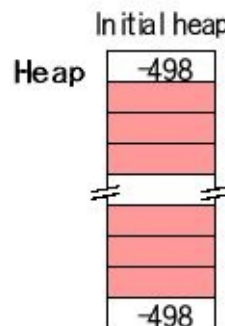
**Checkpoint 1.25:** Using this memory manager, write a malloc and free functions such that the size is restricted to a maximum of 100 bytes. I.e., you may assume the user never asks for more than 100 bytes at a time.

### 1.7.3. Memory manager: malloc and free

The **heap** is a large piece of memory, managed by the operating system, used for temporary allocation. The memory manager has at least three functions: one for initialization (**Heap\_Init**), one function for allocation and a third function for deallocation. Most compilers support memory management, implementing **malloc** and **free**. However, in this example we develop an equivalent solution, with names **Heap\_Malloc** and **Heap\_Free**. You can download a version of the memory manager described in this section at the book web site. It is called **Heap\_xxx** and was developed by Jacob Egner as an example to illustrate programming style. It runs on the TM4C compiled with the ARM Keil uVision, but should operate without change on other microcontrollers and other compilers. The heap itself is statically allocated storage assigned by the compiler. For a 32-bit microcontroller we could define the 2000-byte heap using

```
static int32_t Heap[500];
```

Typically, the operating system calls **Heap\_Init** during the initialization process. The initial heap is one large free block, as shown in Figure 1.43. The initial heap has 498 words of allocatable space and 2 words of overhead.



*Figure 1.43. An initial heap of 2000 bytes is one block of 498 words (each box is 32 bits).*

The proper usage of the dynamic memory manager follows three phases: allocation, use, and deallocation. The user or OS itself calls **Heap\_Malloc** when it needs a contiguous block of memory. It is good design practice to store the pointer to the allocated space in permanent memory. For example, if a 20-byte buffer is needed, initially, we could call

```
int8_t *Pt;  
void UserStart(void){ // called at the beginning  
    Pt = Heap_Malloc(20);  
}
```

The second phase is for the system to use the 20-byte array

```
void UserBody(void){ // called in the middle  
    for(int i=0; i<20; i++){  
        (*Pt) = 0; // access the data via Pt
```

```

    }
    // rest of user programs
}

```

When the program is finished with the block, it is released by calling **Heap\_Free** .

```

void UserFinish(void){ // called at the end
    Heap_Free(Pt);
}

```

**Checkpoint 1.26:** What happens if a function allocates a block, stores a pointer to the block in a local variable, and then returns from the function without deallocating the block?

Saving the pointer to an allocated block in a local variable does not make sense. If the memory is needed for the duration of just one function call, the block should be allocated on the stack. For example, if a 20-byte buffer is needed, we could call

```

void User(void){ int8_t buffer[20];
// use 20-byte buffer
}

```

The heap is divided into blocks of variable size. As shown in Figure 1.44, there are two copies of the block size, one counter stored at the beginning (Header) and other copy of the counter stored at the end of the block (Trailer). These two counters will be classified as internal fragmentation because they exist for the convenience of the operating system. If the counter is positive the block is being used (previously allocated). If the counter is negative the block is free. The value of the counter determines the size of the block in 32-bit words, not including the two counters themselves. If the counter is implemented as a 32-bit signed number ( **int32\_t** ), then a heap of up to  $2^{31} * 4$  bytes (2 gibibytes) can be managed. The number of bytes in a block will be divisible by four. I.e., blocks are aligned to 32-bit word boundaries. For example, if the user asks for a block with 17 bytes, 20 bytes will be allocated. These 3 wasted bytes are a form of **internal fragmentation**. Furthermore, the block with 5 words of data actually requires 7 words of memory.



Figure 1.44. Each block has a header and a trailer.



When allocating blocks we can use a number of algorithms to choose which block to allocate. Let  $n$  be the number of bytes requested by **Heap\_Malloc**.

- **First fit** uses the first free block with a size greater than or equal to  $n$ .
- **Best fit** uses the smallest free block with a size greater than or equal to  $n$ .
- **Worst fit** uses the largest free block with a size greater than or equal to  $n$ .

Depending on the allocation pattern of the user program, these three allocation methods will have differing levels of external fragmentation. The implementation on the book web site as **Heap\_xxx** uses first fit.

**Checkpoint 1.27:** How would you change the way free blocks are organized to implement best fit?

When a block is allocated, a free block is divided to two parts. Figure 1.45 illustrates the process of allocating a 20-word block using a 100-word free block. In this example, 80 bytes is 20 words. The 100-word free block is divided into a 20-word block and a 78-word block. A pointer to the 20-word block is returned by **Heap\_Malloc**.

When allocating a block, the free block may not be large enough to split in two. For example, if the user were to have asked for 392 bytes (98 words) in Figure 1.45, it would be better to give the user the entire 100-word block, because the 8 bytes (2 words) are too small to create a useful block. These extra 8 bytes allocated to the user constitute internal fragmentation.

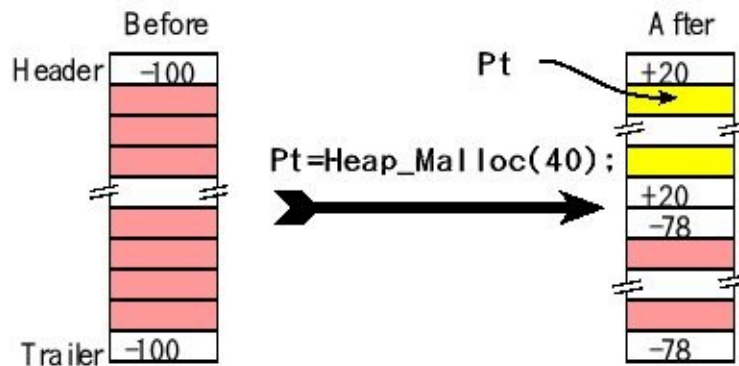


Figure 1.45. Example, the user calls  $Pt = \text{Heap\_Malloc}(80)$ .

**Checkpoint 1.28:** In Figure 1.45, why does the sum of the parts not equal the whole? In particular,  $20 + 78$  does not equal 100.

When deallocating a block, there are four cases: no merge, merge above, merge below and merge both above and below. If the blocks immediately above and immediately below the deallocated block are used, no merging is needed and the manager simply changes the counters from positive to negative, as shown Figure 1.46.

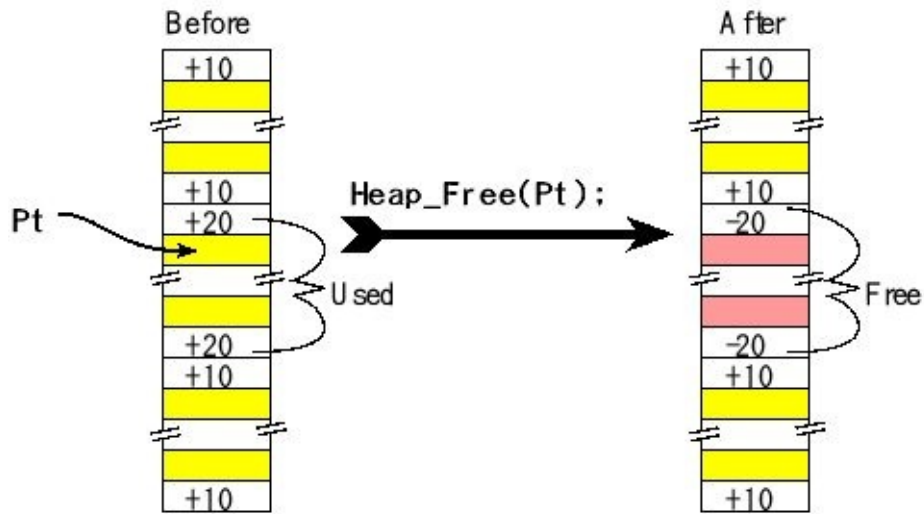


Figure 1.46. Example, the user calls `Heap_Free(Pt)`.

If the block immediately above is free and immediately below is used, a merge above is needed and the manager will combine two blocks into one big free block, as shown Figure 1.47. There are two special cases when deallocating blocks. If the block is the first block in the heap, you cannot merge it above, and if the block is the last block in the heap, you cannot merge it below.

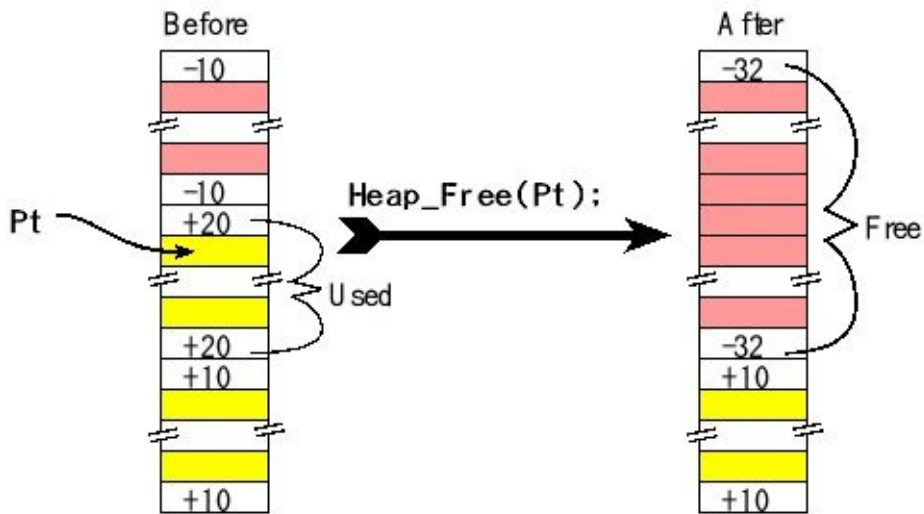


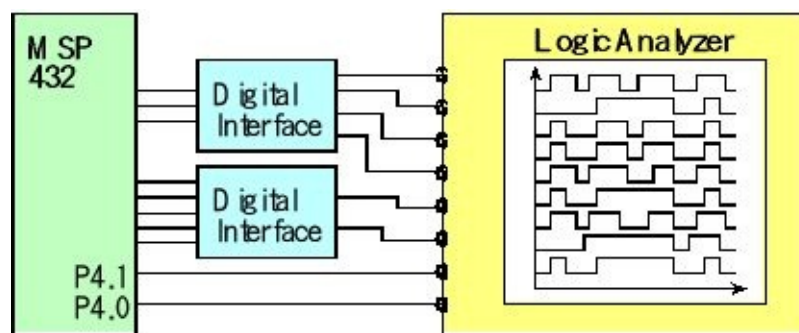
Figure 1.47. Two blocks are merged during a call to `Heap_Free`.

**Checkpoint 1.29:** What happens if you continue to access a memory block after the block is deallocated?

The Knuth **buddy allocation** maintains the heap as a collection of blocks each with a size of  $2^m$ . When the user requests a block of size  $n$ , it will find the smallest block with  $2^m$  greater than or equal to  $n$ . For example, if the smallest block is size 1024, and the user requests a block of 100 bytes, the 1024-byte block will be divided into two 128-byte blocks, one 256-byte block and one 512-byte blocks. The user will be given the 128-byte block. The 28 extra bytes allocated to the user is internal fragmentation.

## 1.8. Introduction to debugging

Microcontroller-related problems often require the use of specialized equipment to debug the system hardware and software. Useful hardware tools include a logic probe, an oscilloscope, a logic analyzer, and a JTAG debugger. A **logic probe** is a handheld device with an LED or buzzer. You place the probe on your digital circuit and LED/buzzer will indicate whether the signal is high or low. An **oscilloscope**, or scope, graphically displays information about an electronic circuit, where the voltage amplitude versus time is displayed. A scope has one or two channels, with many ways to trigger or capture data. A scope is particularly useful when interfacing analog signals using an ADC or DAC. The PicoScope 2104 (from <http://www.picotech.com/>) is a low-cost but effective tool for debugging microcontroller circuits. A **logic analyzer** is essentially a multiple channel digital storage scope with many ways to trigger. As shown in Figure 1.48, we can connect the logic analyzer to digital signals that are part of the system, or we can connect the logic analyzer channels to unused microcontroller pins and add software to toggle those pins at strategic times/places. As a troubleshooting aid, it allows the experimenter to observe numerous digital signals at various points in time and thus make decisions based upon such observations. One problem with logic analyzers is the massive amount of information that it generates. To use an analyzer effectively one must learn proper triggering mechanisms to capture data at appropriate times eliminating the need to sift through volumes of output. The logic analyzer figures in this book were collected with a logic analyzer Digilent (from <http://www.digilentinc.com/>). The Analog Discovery combines a logic analyzer with an oscilloscope, creating an extremely effective debugging tool.



*Figure 1.48. A logic analyzer and example output. P4.1 and P4.0 are extra pins just used for debugging.*

Figure 1.49 shows a logic analyzer output, where signals SSI are outputs to the LCD, and UART is transmission between two microcontrollers. However P3.3 and P3.1 are debugging outputs to measuring timing relationships between software execution and digital I/O. The rising edge of P3.1 is used to trigger the data collection.

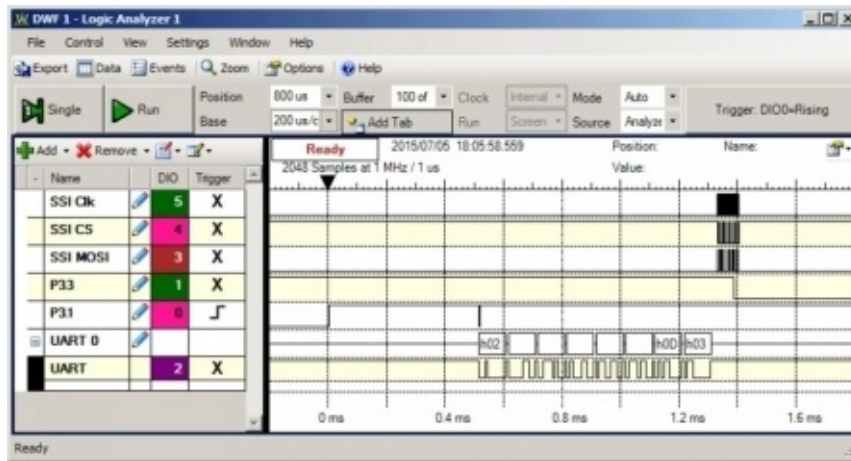


Figure 1.49. Analog Discovery logic analyzer output ([www.digilentinc.com](http://www.digilentinc.com)).

An emulator is a hardware debugging tool that recreates the input/output signals of the processor chip. To use an emulator, we remove the processor chip and insert the emulator cable into the chip socket. In most cases, the emulator/computer system operates at full speed. The emulator allows the programmer to observe and modify internal registers of the processor. Emulators are often integrated into a personal computer, so that its editor, hard drive, and printer are available for the debugging process.

The only disadvantage of the in-circuit emulator is its cost. To provide some of the benefits of this high-priced debugging equipment, many microcontrollers use a JTAG debugger. The JTAG hardware exists both on the microcontroller chip itself and as an external interface to a personal computer. Although not as flexible as an ICE, JTAG can provide the ability to observe software execution in real-time, the ability to set breakpoints, the ability to stop the computer, and the ability to read and write registers, I/O ports and memory.

Debugging is an essential component of embedded system design. We need to consider debugging during all phases of the design cycle. It is important to develop a structure or method when verifying system performance. This section will present a number of tools we can use when debugging. Terms such as program testing, diagnostics, performance debugging, functional debugging, tracing, profiling, instrumentation, visualization, optimization, verification, performance measurement, and execution measurement have specialized meanings, but they are also used interchangeably, and they often describe overlapping functions. For example, the terms profiling, tracing, performance measurement, or execution measurement may be used to describe the process of examining a program from a time viewpoint. But, tracing is also a term that may be used to describe the process of monitoring a program state or history for functional errors, or to describe the process of stepping through a program with a debugger. Usage of these terms among researchers and users vary.

**Black-box testing** is simply observing the inputs and outputs without looking inside. Black-box testing has an important place in debugging a module for its functionality.

On the other hand, **white-box testing** allows you to control and observe the internal workings of a system. A common mistake made by new engineers is to just perform black box testing. Effective debugging uses both. One must always start with black-box testing by subjecting a hardware or software module to appropriate test-cases. Once we document the failed test-cases, we can use them to aid us in effectively performing the task of white-box testing. **Unit testing** involves evaluating each module separately before combining the components into the larger system. **Integration testing** occurs when multiple components are integrated together.

We define a **debugging instrument** as software code that is added to the program for the purpose of debugging. A print statement is a common example of an instrument. Using the editor, we add print statements to our code that either verify proper operation or display run-time errors.

**Nonintrusiveness** is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. **Intrusiveness** is used as a measure of the degree of perturbation caused in program performance by the debugging instrument itself. Let  $t$  be the time required to execute the instrument, and let  $\Delta t$  be the average time in between executions of the instrument. One quantitative measure of intrusiveness is  $t/\Delta t$ , which is the fraction of available processor time used by the debugger. For example, a print statement added to your source code may be very intrusive because it might significantly affect the real-time interaction of the hardware and software. Observing signals that already exist as part of the system with an oscilloscope or logic analyzer is **nonintrusive**, meaning the presence of the scope/analyzer has no effect on the system being measured. A debugging instrument is classified as **minimally intrusive** if it has a negligible effect on the system being debugged. In a real microcontroller system, breakpoints and single-stepping are also intrusive, because the real hardware continues to change while the software has stopped. When a program interacts with real-time events, the performance can be significantly altered when using intrusive debugging tools. To be effective we must employ nonintrusive or minimally intrusive methods.

**Checkpoint 1.30:** What does it mean for a debugging instrument to be minimally intrusive? Give both a general answer and a specific criterion.

Although, a wide variety of program monitoring and debugging tools are available today, in practice it is found that an overwhelming majority of users either still prefer or rely mainly upon “rough and ready” manual methods for locating and correcting program errors. These methods include desk-checking, dumps, and print statements, with print statements being one of the most popular manual methods. Manual methods are useful because they are readily available, and they are relatively simple to use. But, the usefulness of manual methods is limited: they tend to be highly intrusive, and they do not provide adequate control over repeatability, event selection, or event isolation. A real-time system, where software execution timing is critical, usually cannot be debugged with simple print statements, because the print statement itself will require too much time to execute.

The first step of debugging is to **stabilize** the system. In the debugging context, we stabilize the problem by creating a test routine that fixes (or stabilizes) all the inputs. In this way, we can reproduce the exact inputs over and over again. Once stabilized, if we modify the program, we are sure that the change in our outputs is a function of the modification we made in our software and not due to a change in the input parameters.

**Acceleration** means we will speed up the testing process. When we are testing one module we can increase how fast the functions are called in an attempt to expose possible faults. Furthermore, since we can control the test environment, we will **vary** the test conditions over a wide range of possible conditions. **Stress testing** means we run the system beyond the requirements to see at what point it breaks down.

When a system has a small number of possible inputs (e.g., less than a million), it makes sense to test them all. When the number of possible inputs is large we need to choose a set of inputs. **Coverage** defines the subset of possible inputs selected for testing. A **corner case** is defined as a situation at the boundary where multiple inputs are at their maximum, like the corner of a 3-D cube. At the corner small changes in input may cause lots of internal and external changes. In particular, we need to test the cases we think might be difficult (e.g., the clock output increments one second from 11:59:59 PM December 31, 1999.) There are many ways to decide on the coverage. We can select values:

- Near the extremes and in the middle
- Most typical of how our clients will properly use the system
- Most typical of how our clients will improperly use the system
- That differ by one
- You know your system will find difficult
- Using a random number generator

**Maintenance Tip:** First, find the things that will break you. Second, break them.

To stabilize the system we define a fixed set of inputs to test, run the system on these inputs, and record the outputs. Debugging is a process of finding patterns in the differences between recorded behavior and expected results. The advantage of modular programming is that we can perform **modular debugging**. We make a list of modules that might be causing the bug. We can then create new test routines to stabilize these modules and debug them one at a time. Unfortunately, sometimes all the modules seem to work, but the combination of modules does not. In this case we study the interfaces between the modules, looking for intended and unintended (e.g., unfriendly code) interactions.

**Common error:** Sometimes the original system operates properly, and the debugging code has bugs.

---

## 1.9. Exercises

**1.1** There are two R13s. What is special about R13? Why are there two of them? What is the initial value in R13 after a reset?

**1.2** What is in R14 when a function is called? How do you write code so that function calls can be nested? What is the initial value in R14 after a reset?

**1.3** What is in Register 15? Why is bit 0 of Register 15 always 0? What happens when you load a value into Register 15 with bit 0 set? What is the initial value in R15 after a reset?

**1.4** Why are there so many buses on the ARM Cortex-M processor?

**1.5** Write C code that sets bit 30 of memory location 0x2000.4000 using bit-banding.

**1.6** Write C code that clears bit 15 of memory location 0x2000.1000 using bit-banding.

**1.7** Write C code that sets bit 5 of memory location 0x4000.4400 using bit-banding. What effect does this operation have?

**1.8** Write C code that clears bit 3 of memory location 0x4000.7400 using bit-banding. What effect does this operation have?

**1.9** Where is the interrupt enable bit on ARM Cortex-M processor? Which value enables interrupts: 0 or 1?

**1.10** Does the associative principle hold for signed integer multiply and divide? Assume **Out1** **Out2** **A** **B** **C** are all the same precision (e.g., 32 bits). In particular do these two C calculations always achieve identical outputs? If not, give an example.

**Out1 = (A\*B)/C;                      Out2 = A\*(B/C);**

**1.11** Does the associative principle hold for signed integer addition and subtraction? Assume **Out3** **Out4** **A** **B** **C** are all the same precision (e.g., 32 bits). In particular do these two C calculations always achieve identical outputs? If not, give an example.

**Out3 = (A+B)-C;                      Out4 = A+(B-C);**

**1.12** According to AAPCS, which registers must be preserved and which registers are free to modify by a function?

**1.13** A C function has this prototype, **void MyProg(int a, int b, int c)** . If one placed a breakpoint at the beginning of this function, where would you find the parameters a, b, and c?

**1.14** Write two assembly functions that return R0 equal to 31 times the input. One function uses the multiply function and one uses the shift and reverse subtract. Make

the functions comply with AAPC, so R0 is the input and R0 is the output.

**1.15** Let R0 and R1 be two unsigned integers. Write assembly code that makes R0 the larger of the two using the conditional assembly instruction **IT**.

**1.16** Consider a software system that allocates memory block  $i$  of  $Size_i$  in the order of  $i = 0, 1, 2, \dots$ . In this system, blocks will always be deallocated in the opposite order. Prove that the memory manage will never result in fragmentation (two free blocks that are not adjacent.) Write three functions (init, malloc, and free) that implement a heap used in this manner.

```
#define SIZE 1000  
uint8_t Heap[SIZE];
```



# 2. Microcontroller Input/Output

## Chapter 2 objectives are to:

- Overview digital I/O on the MSP432 and TM4C
- Review interrupt synchronization
- Introduce timer and edge-triggered interrupts
- Define simple serial communication using the UART and SPI
- Present timer I/O with input capture and PWM
- Overview analog I/O using a DAC and an ADC

The overall objective of this book is to teach the design of real-time operating systems for embedded systems. This chapter will review interfacing to the Texas Instruments MSP432/TM4C family of microcontrollers. Hardware and software aspects of interfacing to the microcontroller were presented in detail in Volume 2. In particular, this chapter is an abridged version of Volume 2 summarizing I/O interfacing concepts, presenting some reference material. The reader can refer to Volume 2 for more details including more design examples.

---

## 2.1. Parallel I/O

On most embedded microcontrollers, the I/O ports are **memory mapped**. This means the software can access an input/output port simply by reading from or writing to the appropriate address. It is important to realize that even though I/O operations “look” like reads and writes to memory variables, the I/O ports often DO NOT act like memory. For example, some bits are read-only, some are write-only, some can only be cleared, others can only be set, and some bits cannot be modified. To make our software more readable we include symbolic definitions for the I/O ports. We set the direction register to specify which pins are input and which are output. Individual port pins can be general purpose I/O (GPIO) or have an alternate function.

With a parallel input software reads a binary one if the input pin is high. The software reads a binary zero if the input pin is low. With a parallel output, when the software writes a 1, the output pin goes high. When the software writes a 0, the output pin goes low. Microcontrollers allow parallel I/O to 8 or 16 pins at a time, hence the classification as **parallel I/O**.

### 2.1.1. TM4C I/O programming

Pins have a regular (GPIO) or can have one of multiple alternate functions. By default, the alternate function register (e.g., **GPIO\_PORTD\_AFSEL\_R**) is zero, specifying the corresponding bits are regular GPIO pins. We will set bits in the alternative function register when we wish to activate the functions listed in Tables 1.4, and 1.5. Typically, we write to the direction and alternate function registers once during the initialization phase. We use the data register (e.g., **GPIO\_PORTD\_DATA\_R**) to perform input/output on the port. Conversely, we read and write the data register multiple times to perform input and output respectively during the running phase. The only differences among the TM4C family are the number of ports and available pins in each port. For example, the TM4C1294 has fifteen digital I/O ports A (8 bits), B (6 bits), C (8 bits), D (8 bits), E (6 bits), F (5 bits), G (2 bits), H (4 bits), J (2 bits), K (8 bits), L (8 bits), M (8 bits), N (6 bits), P (6 bits), and Q (5 bits). Furthermore, the TM4C1294 has different addresses for ports. Refer to the file **tm4c1294ncpdt.h** or to the data sheet for more the specific addresses of its I/O ports.

To initialize an I/O port for general use we perform seven steps, see Program 2.1. We will skip steps three four and six in this chapter because the default state after a reset is to disable analog function and disable alternate function. First, we activate the clock for the port by setting the corresponding bit in **RCGCGPIO** register. Because it takes time for the clock to stabilize, we next will wait for its status bit in the **PRGPIO** to be true. Second, we unlock the port; unlocking is needed only for pins

PD7, and PF0 on the TM4C123. The only pin needing unlocking on the TM4C1294 is PD7. Third, we disable the analog function of the pin, because we will be using the pin for digital I/O. Fourth, we clear bits in the **PCTL** (Tables 1.4, 1.5) to select regular digital function. Fifth, we set its direction register. The direction register specifies bit for bit whether the corresponding pins are input or output. A bit in **DIR** set to 0 means input and 1 means output. Sixth, we clear bits in the alternate function register, and lastly, we enable the digital port. Turning on the clock must be first. If the pin needs unlocking that must be second. However, the other five steps can occur in any order.

```

void PortF_Init(void){ // TM4C123 has PortF bits 4-0
SYSCTL_RCGCGPIO_R |= 0x00000020; // 1) activate clock for Port F
while((SYSCTL_PRGPIO_R&0x00000020) == 0){}; // wait for stabilization
GPIO_PORTF_LOCK_R = 0x4C4F434B; // 2) unlock GPIO Port F
GPIO_PORTF_CR_R = 0x1F; // allow changes to PF4-0
GPIO_PORTF_AMSEL_R = 0x00; // 3) disable analog on PF
GPIO_PORTF_PCTL_R = 0x00000000; // 4) PCTL GPIO on PF4-0
GPIO_PORTF_DIR_R = 0x0E; // 5) PF4,PF0 in, PF3-1 out
GPIO_PORTF_AFSEL_R = 0x00; // 6) disable alt funct on PF4-0
GPIO_PORTF_PUR_R = 0x11; // enable pull-up on PF0 and PF4
GPIO_PORTF_DEN_R = 0x1F; // 7) enable digital I/O on PF4-0
}
uint32_t PortF_Input(void){
return (GPIO_PORTF_DATA_R&0x11); // read PF4,PF0 inputs
}
void PortF_Output(uint32_t data){
GPIO_PORTF_DATA_R = data; // write PF3-PF1 outputs
}

```

*Program 2.1. A set of functions using PF4, PF0 as inputs and PF3–PF1 as outputs.*

Address	7	6	5	4	3	2	1	0	Name
\$400F.E608	-	-	GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_RCGCGPIO_R
\$4000.43FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTA_DATA_R
\$4000.4400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTA_DIR_R
\$4000.4420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTA_AFSEL_R
\$4000.4510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTA_PUR_R
\$4000.451C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTA_DEN_R
\$4000.4524	1	1	1	1	1	1	1	1	GPIO_PORTA_CR_R
\$4000.4528	0	0	0	0	0	0	0	0	GPIO_PORTA_AMSEL_R
\$4000.53FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTB_DATA_R
\$4000.5400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTB_DIR_R
\$4000.5420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTB_AFSEL_R
\$4000.5510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTB_PUR_R
\$4000.551C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTB_DEN_R
\$4000.5524	1	1	1	1	1	1	1	1	GPIO_PORTB_CR_R

\$4000.5528	0	0	AMSEL	AMSEL	0	0	0	0	GPIO_PORTB_AMSEL_R
\$4000.63FC	DATA	DATA	DATA	DATA	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_DATA_R
\$4000.6400	DIR	DIR	DIR	DIR	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_DIR_R
\$4000.6420	SEL	SEL	SEL	SEL	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_AFSEL_R
\$4000.6510	PUE	PUE	PUE	PUE	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_PUR_R
\$4000.651C	DEN	DEN	DEN	DEN	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_DEN_R
\$4000.6524	1	1	1	1	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_CR_R
\$4000.6528	AMSEL	AMSEL	AMSEL	AMSEL	JTAG	JTAG	JTAG	JTAG	GPIO_PORTC_AMSEL_R
\$4000.73FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTD_DATA_R
\$4000.7400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTD_DIR_R
\$4000.7420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTD_AFSEL_R
\$4000.7510	PUE	PUE	PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTD_PUR_R
\$4000.751C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTD_DEN_R
\$4000.7524	CR	1	1	1	1	1	1	1	GPIO_PORTD_CR_R
\$4000.7528	0	0	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTD_AMSEL_R
\$4002.43FC			DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTE_DATA_R
\$4002.4400			DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTE_DIR_R
\$4002.4420			SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTE_AFSEL_R
\$4002.4510			PUE	PUE	PUE	PUE	PUE	PUE	GPIO_PORTE_PUR_R
\$4002.451C			DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTE_DEN_R
\$4002.4524			1	1	1	1	1	1	GPIO_PORTE_CR_R
\$4002.4528			AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTE_AMSEL_R
\$4002.53FC				DATA	DATA	DATA	DATA	DATA	GPIO_PORTF_DATA_R
\$4002.5400				DIR	DIR	DIR	DIR	DIR	GPIO_PORTF_DIR_R
\$4002.5420				SEL	SEL	SEL	SEL	SEL	GPIO_PORTF_AFSEL_R
\$4002.5510				PUE	PUE	PUE	PUE	PUE	GPIO_PORTF_PUR_R
\$4002.551C				DEN	DEN	DEN	DEN	DEN	GPIO_PORTF_DEN_R
\$4002.5524				1	1	1	1	CR	GPIO_PORTF_CR_R
\$4002.5528				0	0	0	0	0	GPIO_PORTF_AMSEL_R
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0	
\$4000.452C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTA_PCTL_R
\$4000.552C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTB_PCTL_R
\$4000.652C	PMC7	PMC6	PMC5	PMC4	0x1	0x1	0x1	0x1	GPIO_PORTC_PCTL_R
\$4000.752C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTD_PCTL_R
\$4002.452C			PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTE_PCTL_R
\$4002.552C				PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTF_PCTL_R
\$4000.6520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO_PORTC_LOCK_R
\$4000.7520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO_PORTD_LOCK_R
\$4002.5520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO_PORTF_LOCK_R

**Table 2.1. Some TM4C123 parallel ports. Each register is 32 bits wide. For PMCx bits, see Tables 1.4 and 1.5. JTAG means do not use these pins and do not change any of these bits.**

To use a port we first must activate its clock in the **SYSCTL\_RCGCGPIO\_R** register. To make Port F pins 4,0 input and pins 3–1 output, we set the direction register to 0x0E, as shown in Program 2.1. When the software reads from location 0x400253FC the bottom 5 bits are returned with the current values on Port F. The top 27 bits are returned zero. The input pins show the current digital state, and the output pins show the value last written to the port. The function **PortF\_Input** will read from the two input pins and return a value, 0x00

0x01, 0x10 or 0x11, depending on the current status of the inputs. The function **PortF\_Output** will write new values to the three output pins.

In Program 2.1 the assumption was the software module had access to all of Port F. In other words, this software owned all five pins of Port F. In most cases, a software module needs access to only some of the port pins. If two or more software modules access the same port, a conflict will occur if one module changes modes or output values owned by another module. It is good software design to write **friendly** software, which only affects the individual pins as needed. Friendly software does not change the other bits in a shared register. Conversely, **unfriendly** software modifies more bits of a register than it needs to. The difficulty of unfriendly code is each module will run properly when tested by itself, but weird bugs result when two or more modules are combined.

Consider the problem that a software module need to output to just Port F bit 1. After enabling the clock for Port F, we use read-modify-write software to initialize just pin 1

```
SYSCTL_RCGCGPIO_R |= 0x00000020; // activate clock for Port F  
while((SYSCTL_PRGPIO_R&0x00000020) == 0){}; // clock stabilization  
GPIO_PORTF_AMSEL_R &= ~0x02; // disable analog on PF1  
GPIO_PORTF_PCTL_R &= ~0x000000F0; // PCTL GPIO on PF1  
GPIO_PORTF_DIR_R |= 0x02; // PF1 is an output  
GPIO_PORTF_AFSEL_R &= ~0x02; // regular port function  
GPIO_PORTF_DEN_R |= 0x02; // PF1 is enabled as a digital port
```

There is no conflict if two or more modules enable the clock for Port F. There are two ways on the Cortex-M microcontroller to access individual port bits. The first method is to use read-modify-write software to change just pin 1. A read-or-write sequence can be used to set one or more bits.

```
GPIO_PORTF_DATA_R |= 0x02; // make PF1 high
```

A read-and-write sequence can be used to clear one or more bits.

```
GPIO_PORTF_DATA_R &= ~0x02; // make PF1 low
```

The second method uses the bit-specific addressing. The TM4C family implements a more flexible way to access port pins than the bit-banding. This bit-specific addressing doesn't work for all the I/O registers, just the parallel port data registers. This mechanism allows collective access to 1 to 8 bits in a data port. We define eight address offset constants in Table 2.2. Basically, if we are interested in bit  $b$ , the constant is  $4*2^b$ . There 256 possible bit combinations we might be interested in accessing, from all of them to none of them. Each possible bit combination has a separate address for accessing that combination. For each bit we are interested in, we add up the corresponding constants from Table 2.2 and then add that sum to the base address for the port. The base addresses for the data ports can be found in GPIO

chapter of the microcontroller data sheet. For example, assume we are interested in Port A bits 1, 2, and 3. The base address for Port A is 0x4000.4000, and the constants are 0x0020, 0x0010 and 0x0008. The sum of 0x4000.4000+0x0020+0x0010+0x0008 is the address 0x4000.4038. If we read from 0x4000.4038 only bits 1, 2, and 3 will be returned. If we write to this address only bits 1, 2, and 3 will be modified.

<i>If we wish to access bit</i>	<i>Constant</i>
7	0x0200
6	0x0100
5	0x0080
4	0x0040
3	0x0020
2	0x0010
1	0x0008
0	0x0004

**Table 2.2. Address offsets used to specify individual data port bits.**

The base address for Port F is 0x4002.5000. If we want to read and write all 8 bits of this port, the constants will add up to 0x03FC. Notice that the sum of the base address and the constants yields the 0x4002.53FC address used in **tm4c123gh6pm.h**. In other words, read and write operations to 0x4002.53FC will access all bits of Port F. If we are interested in just bit 1 of Port F, we add 0x0008 to 0x4002.5000, to get 0x4002.5008. Now, a simple write operation can be used to set **PF1**. The following macros are friendly because it does not modify the other bits of Port F. A read from **PF1** will return 0x02 or 0x00 depending on whether the pin is high or low, respectively. The PF1 and PF2 macros are not critical with respect each other.

```
#define PF1 (*(volatile uint32_t *)0x40025008)
#define SetPF1() (PF1 = 0x02)
#define ClearPF1() (PF1 = 0x00)
    #define TogglePF1()(PF1 = PF1^0x02)
#define PF2 (*(volatile uint32_t *)0x40025010)
#define SetPF2() (PF2 = 0x04)
#define ClearPF2() (PF2 = 0x00)
    #define TogglePF2() (PF2 = PF2^0x04)
```

## 2.1.2. MSP432 I/O programming

We will set/clear bits in the select registers (e.g., **P1SEL1 P1SEL0**) when we wish to activate the alternate functions listed in Table 2.3. To use a pin as GPIO, we must clear the corresponding bits in the two select registers. Typically, we write to the direction and select registers once during the initialization phase. We use the data

registers(e.g., **P1IN P1OUT** ) to perform the actual input/output on the port. Table 2.4 shows the parallel port registers for Ports 1 and 2, but there are similar registers for other ports 3 – 10. Each register in Table 2.4 is 8 bits wide.

To make a pin an output, we set the corresponding bit in the **PxDIR** register to 1. In addition, we can also set the corresponding bit in the drive strength register (e.g., **P2DS** ) to increase the maximum  $I_{OL}$  and  $I_{OH}$  of the pin to 20 mA. Normal strength is **DS=0**, and increased strength, called high drive, is **DS=1**. High-drive with **DS=1** is available only on P2.0 – P2.3.

Pin	PxSEL1=0, PxSEL0=0	PxSEL1=0, PxSEL0=1	PxSEL1=1, PxSEL0=0	PxSEL1=1, PxSEL0=1
P1.0	Port	UCA0STE		
P1.1	Port	UCA0CLK		
P1.2	Port	UCA0RXD/UCA0SOMI		
P1.3	Port	UCA0TXD/UCA0SIMO		
P1.4	Port	UCB0STE		
P1.5	Port	UCB0CLK		
P1.6	Port	UCB0SIMO/UCB0SDA		
P1.7	Port	UCB0SOMI/UCB0SCL		
P2.0	Port	UCA1STE		
P2.1	Port	UCA1CLK		
P2.2	Port	UCA1RXD/UCA1SOMI		
P2.3	Port	UCA1TXD/UCA1SIMO		
P2.4	Port	TA0.CCI1A <sup>a</sup> / TA0.1 <sup>b</sup>		
P2.5	Port	TA0.CCI2A <sup>a</sup> / TA0.2 <sup>b</sup>		
P2.6	Port	TA0.CCI3A <sup>a</sup> / TA0.3 <sup>b</sup>		
P2.7	Port	TA0.CCI4A <sup>a</sup> / TA0.4 <sup>b</sup>		
P3.0	Port	UCA2STE		
P3.1	Port	UCA2CLK		
P3.2	Port	UCA2RXD/UCA2SOMI		
P3.3	Port	UCA2TXD/UCA2SIMO		
P3.4	Port	UCB2STE		
P3.5	Port	UCB2CLK		
P3.6	Port	UCB2SIMO/UCB2SDA		
P3.7	Port	UCB2SOMI/UCB2SCL		
P4.0	Port			A13
P4.1	Port			A12
P4.2	Port	ACLK <sup>b</sup>	TA2CLK <sup>a</sup>	A11
P4.3	Port	MCLK <sup>b</sup>	RTCCLK <sup>b</sup>	A10
P4.4	Port	HSMCLK <sup>b</sup>	SVMHOUT <sup>b</sup>	A9
P4.5	Port			A8
P4.6	Port			A7
P4.7	Port			A6
P5.0	Port			A5
P5.1	Port			A4
P5.2	Port			A3
P5.3	Port			A2
P5.4	Port			A1

P5.5	Port			A0
P5.6	Port	TA2.CCI1A <sup>a</sup> / TA2.1 <sup>b</sup>		VREF+, VeREF+, C1.7
P5.7	Port	TA2.CCI2A <sup>a</sup> / TA2.2 <sup>b</sup>		VREF-, VeREF-, C1.6
P6.0	Port			A15
P6.1	Port			A14
P6.2	Port	UCB1STE		C1.5
P6.3	Port	UCB1CLK		C1.4
P6.4	Port	UCB1SIMO/UCB1SDA		C1.3
P6.5	Port	UCB1SOMI/UCB1SCL		C1.2
P6.6	Port	TA2.CCI3A <sup>a</sup> / TA2.3 <sup>b</sup>	UCB3SIMO/UCB3SDA	C1.1
P6.7	Port	TA2.CCI4A <sup>a</sup> / TA2.4 <sup>b</sup>	UCB3SOMI/UCB3SCL	C1.0
P7.0	Port	DMAE0 <sup>a</sup> / SMCLK <sup>b</sup>		
P7.1	Port	TA0CLK <sup>a</sup> / C0OUT <sup>b</sup>		
P7.2	Port	TA1CLK <sup>a</sup> / C1OUT <sup>b</sup>		
P7.3	Port	TA0.CCI0A <sup>a</sup> / TA0.0 <sup>b</sup>		
P7.4	Port	TA1.CCI4A <sup>a</sup> / TA1.4 <sup>b</sup>		C0.5
P7.5	Port	TA1.CCI3A <sup>a</sup> / TA1.3 <sup>b</sup>		C0.4
P7.6	Port	TA1.CCI2A <sup>a</sup> / TA1.2 <sup>b</sup>		C0.3
P7.7	Port	TA1.CCI1A <sup>a</sup> / TA1.1 <sup>b</sup>		C0.2
P8.0	Port	UCB3STE	TA1.CCI0A <sup>a</sup> / TA1.0 <sup>b</sup>	C0.1
P8.1	Port	UCB3CLK	TA2.CCI0A <sup>a</sup> / TA2.0 <sup>b</sup>	C0.0
<b>Pin</b>	<b>PxSEL1=0, PxSEL0=0</b>	<b>PxSEL1=0, PxSEL0=1</b>	<b>PxSEL1=1, PxSEL0=0</b>	<b>PxSEL1=1, PxSEL0=1</b>
P8.2	Port	TA3.CCI2A <sup>a</sup> / TA3.2 <sup>b</sup>		A23
P8.3	Port	TA3CLK <sup>a</sup>		A22
P8.4	Port			A21
P8.5	Port			A20
P8.6	Port			A19
P8.7	Port			A18
P9.0	Port			A17
P9.1	Port			A16
P9.2	Port	TA3.CCI3A <sup>a</sup> / TA3.3 <sup>b</sup>		
P9.3	Port	TA3.CCI4A <sup>a</sup> / TA3.4 <sup>b</sup>		
P9.4	Port	UCA3STE		
P9.5	Port	UCA3CLK		
P9.6	Port	UCA3RXD/UCA3SOMI		
P9.7	Port	UCA3TXD/UCA3SIMO		
P10.0	Port	UCB3STE		
P10.1	Port	UCB3CLK		
P10.2	Port	UCB3SIMO/UCB3SDA		
P10.3	Port	UCB3SOMI/UCB3SCL		
P10.4	Port	TA3.CCI0A <sup>a</sup> / TA3.0 <sup>b</sup>		C0.7
P10.5	Port	TA3.CCI1A <sup>a</sup> / TA3.1 <sup>b</sup>		C0.6

**Table 2.3. SEL1 and SEL0 bits on the MSP432 specify alternate functions. P1.2 and P1.3**



are hardwired to the serial port. <sup>a</sup> means DIR register is zero, <sup>b</sup> means DIR register is one.

To make a pin an input, we clear the corresponding bit in the **PxDIR** register to 0. In addition, we can activate a pull up or pull down resistor on an input pin. To activate a pull up resistor, we set **PxREN**=1 and **PxOUT**=1. To activate a pull down resistor, we set **PxREN**=1 and clear **PxOUT**=0. The equivalent resistance of the pull up or pull down resistor is about 20 – 50 kΩ.

Address	7	6	5	4	3	2	1	0	Name
0x4000.4C00	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	P1IN
0x4000.4C02	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	P1OUT
0x4000.4C04	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	P1DIR
0x4000.4C06	REN	REN	REN	REN	REN	REN	REN	REN	P1REN
0x4000.4C08	DS	DS	DS	DS	DS	DS	DS	DS	P1DS
0x4000.4C0A	SEL0	SEL0	SEL0	SEL0	SEL0	SEL0	SEL0	SEL0	P1SEL0
0x4000.4C0C	SEL1	SEL1	SEL1	SEL1	SEL1	SEL1	SEL1	SEL1	P1SEL1
0x4000.4C01	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	P2IN
0x4000.4C03	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	P2OUT
0x4000.4C05	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	P2DIR
0x4000.4C07	REN	REN	REN	REN	REN	REN	REN	REN	P2REN
0x4000.4C09	DS	DS	DS	DS	DS	DS	DS	DS	P2DS
0x4000.4C0B	SEL0	SEL0	SEL0	SEL0	SEL0	SEL0	SEL0	SEL0	P2SEL0
0x4000.4C0D	SEL1	SEL1	SEL1	SEL1	SEL1	SEL1	SEL1	SEL1	P2SEL1

**Table 2.4. MSP432 parallel ports 1 and 2. Each register is 8 bits wide. For SEL bits, see Table 2.3.**

Table 2.5 lists the possible ways to configure a GPIO pin. To initialize an I/O port for general use we perform three steps. First, we specify GPIO writing zeros to the **PxSEL0** and **PxSEL1** registers. Second, we set its direction register. The direction register specifies bit for bit whether the corresponding pins are input or output, 0 means input and 1 means output. Third, for inputs we can add a pull up or pull down resistor. For outputs we can specify drive strength using **P2DS** on P2.0 – P2.3.

<b>PxDIR</b>	<b>PxOut</b>	<b>PxDS</b>	<b>PxREN</b>	<b>Functionality</b>
0	X	X	0	Regular GPIO input
0	0	X	1	GPIO input with pull down
0	1	X	1	GPIO input with pull up
1	0	0	X	Regular GPIO output low
1	1	0	X	Regular GPIO output high
1	0	0	1	High drive GPIO

				output low
1	1	0	1	High drive GPIO output high

**Table 2.5. MSP432 GPIO functions, assuming PxSEL0 and PxSEL1 are zero. The little x specifies port 1 – 10. The big X means don't care.**

A 16-bit read access from address 0x40004C00 (defined as **PAIN**) will return the input values from both Ports 1 and 2 as one 16-bit result. Since the ARM is little endian, Port 1 will be in least significant bits and Port 2 will be in the most significant bits. Similarly, a 16-bit write access to address 0x40004C02 (defined as **PAOUT**) will set the output values to both Ports 1 and 2 in one 16-bit operation. In fact, we have 16-bit names for each set of adjacent 8-bit ports. 16-bit port definitions are available for Ports A – E. Definitions for Port A are shown below.

```
#define PAIN (HWREG16(0x40004C00)) // Input
#define PAOUT (HWREG16(0x40004C02)) // Output
#define PADIR (HWREG16(0x40004C04)) // Direction
#define PAREN (HWREG16(0x40004C06)) // Resistor
#define PADS (HWREG16(0x40004C08)) // Strength
#define PASEL0 (HWREG16(0x40004C0A)) // Select 0
#define PASEL1 (HWREG16(0x40004C0C)) // Select 1
```

Port A is	Port 2	: Port 1
Port B is	Port 4	: Port 3
Port C is	Port 6	: Port 5
Port D is	Port 8	: Port 7
Port E is	Port 10	: Port 9

In this first example, we will initialize the LaunchPad so we can read from the two switches and output to the 3-color LED. In particular, we will make P1.4 and P1.1 GPIO inputs, and we will make P2.2-P2.0 GPIO outputs, as shown in Program 2.2. To run this example on the LaunchPad, we also set bits in the **P1REN** register for the two switch inputs to have an internal pull-up resistor, equivalent to 20 – 50 kΩ. To make the resistor a pull up to 3.3V, the initialization software sets the corresponding bits in the **P1OUT** register.

When the software performs an 8-bit read from location 0x40004C00, the 8 bits are

returned with the values currently on Port 1. When reading an I/O port, the input pins report the high/low state currently on the input, and the output pins show the value last written to the port. The function **Port1\_Input** will read from all eight Port 1 pins, and return a value depending on the status of the pins at the time of the read.

When writing to an I/O port, the input pins are not affected, and the output pins are changed to the value written to the port. That value remains until written again. The function **Port2\_Output** will write new values to the output pins. The **#include** will define symbolic names for all the I/O ports for that microcontroller. The **mcp432p401r.h** file comes with the compiler installation. Use the proper one for your microcontroller. Program 2.2 writes all bits of the port registers, and this is an inappropriate method of I/O programming. In general, it is better to set/clear bits on an individual basis.

**Observation:** High drive strength (DS=1) is only available on P2.0 P2.1 P2.2 and P2.3. Setting DS=1 does not make the current 20 mA, rather makes it possible for the pin to drive up to 20 mA if needed.

```
void Port1_Init(void){
    P1SEL0 &= ~0x12;
    P1SEL1 &= ~0x12; // 1) configure P1.4 and P1.1 as GPIO
    P1DIR &= ~0x12; // 2) make P1.4 and P1.1 in
    P1REN |= 0x12; // 3) enable pull resistors on P1.4 and P1.1
    P1OUT |= 0x12; // P1.4 and P1.1 are pull-up
}
uint8_t Port1_Input(void){
    return (P1IN&0x12); // read P1.4,P1.1 inputs
}
void Port2_Init(void){
    P2SEL0 &= ~0x07;
    P2SEL1 &= ~0x07; // 1) configure P2.2-P2.0 as GPIO
    P2DIR |= 0x07; // 2) make P2.2-P2.0 out
    P2DS |= 0x07; // 3) activate increased drive strength
    P2OUT &= ~0x07; // all LEDs off
}
void Port2_Output(uint8_t data){ // write three outputs bits of P2
    P2OUT = (P2OUT&0xF8)|data;
}
```

*Program 2.2. A set of functions using P1.4,P1.1 as inputs and P2.2-0 as outputs (InputOutput\_MSP432).*

**Checkpoint 2.1:** Does the entire port need to be defined as input or output, or can some pins be input while others are output?

In Program 2.2 the assumption was the software module did not have access to all of Port 2. In other words, this software owned only P1.4, P1.1, P2.2, P2.1, and P2.0.

Good design practice clearly specifies which pins belong to which software modules. If two or more software modules access the same port, a conflict will occur if one module changes modes or output values owned by another module. It is good software design to write **friendly** software, which only affects the individual pins as needed. Friendly software does not change the other bits in a shared register. Conversely, **unfriendly** software modifies more bits of a register than it needs to. The difficulty of unfriendly code is each module will run properly when tested by itself, but weird bugs result when two or more modules are combined. A read-or-write sequence can be used to set one or more bits. A read-and-write sequence can be used to clear one or more bits.

The second method uses the bit-banding. In this example, assume P1.0 is an output connected to the LED. The regular 8-bit access for **P1OUT** is 0x40004C02. For bit-banding of bit 0 of this address,  $n=0x4C01$  and  $b=0$ . The address for this bit will be

$$0x42000000 + 32*n + 4*b = 0x42000000 + 32*0x4C02 + 4*0 = 0x42098040$$

In C we can create an I/O port label for just bit 0 of Port 1 output.

```
#define LEDOUT (*((volatile uint8_t *) (0x42098040)))
```

With this bit-banded definition, accessing P1.0 is much simpler. Writing a 1 to a bit-banded address sets that bit, and writing a 0 clears that bit (without affecting the other 7 bits).

```
#define LED_On() (LEDOUT = 0x01)
```

```
#define LED_Off() (LEDOUT = 0x00)
```

We can also create bit-banded addresses for the two switches on the LaunchPad. Reading a bit-banded address returns 0 or 1 depending on if the bit is clear or set. SW2 is Port 1 bit 4 and SW1 is Port 1 bit 1. The address of **P1IN** is 0x40004C00. For bit-banding of this address,  $n=0x4C00$ . The aliased addresses for bits 4 and 1 will be

$$0x42000000 + 32*0x4C00 + 4*4 = 0x42098010$$

$$0x42000000 + 32*0x4C00 + 4*1 = 0x42098004$$

In C we can create I/O port label for SW1 and SW2 input.

```
#define SW2IN (*((volatile uint8_t *) (0x42098010)))
```

```
#define SW1IN (*((volatile uint8_t *) (0x42098004)))
```

The switches are negative logic. Using **SW2IN** will return a 1 if P1.4 is 1 (SW2 switch not pressed), and will return a 0 if P1.4 is 0 (SW2 switch pressed). Using **SW1IN** will return a 1 if P1.1 is 1 (SW1 switch not pressed), and will return a 0 if P1.1 is 0 (SW1 switch pressed).

Bit-banding only works for individual bits. It cannot be used to access more than one bit at a time. Recall the 3-color LED is interfaced on P2.2 P2.1 and P2.0. There is no bit-banded address to allow us to set all three bits in one operation. We could use bit-banding to access the colors on P2.2 P2.1 and P2.0 individually.

```
#define BLUELED (*((volatile uint8_t *)0x42098068))  
#define GREENLED (*((volatile uint8_t *)0x42098064))  
#define REDLED (*((volatile uint8_t *)0x42098060))
```

To make the LED yellow, we turn on red, turn on green, and turn off blue:

```
REDLED = 1;  
GREENLED = 1;  
BLUELED = 0;
```

## 2.2. Interrupts

Another concept we need the reader to have a thorough understanding of is an Interrupt. An **interrupt** is a hardware/software triggered software action, see Figure 2.1. In this class we will see three types of interrupts. A software interrupt is triggered by software. Executing the **SVC** (supervisor call) instruction will generate an interrupt. There is another software interrupt on the Cortex M called **PendSV**, which is also triggered by software. We will see a third mechanism for software interrupt in this chapter where the software executes explicit code to trigger a SysTick timer interrupt.

The second type of interrupt is a periodic interrupt, which is triggered periodically by a hardware timer. The MSP432/TM4C microcontrollers have SysTick and Timer interrupts. The ISR will perform an action we wish to perform on a regular basis. For example, a data acquisition system needs to read the ADC at a regular rate.

The third type of interrupt is triggered by input/output events. With an input device, the hardware will request an interrupt when input device has new data. The software interrupt service routine (ISR) will read from the input device and save (put) the data into a data structure located in shared memory, see Figure 2.1. When the system wishes to process the data, it will check the status of the data structure, and if there is some data it will get it from the data structure located in shared memory.

With an output device, the hardware will request an interrupt when the output device is idle. The ISR will get data from a data structure located in shared memory, and then write to the device. When the system wishes to output data, it will check the status of the data structure, and if there is room in the data structure, software will write (put) its data.

Interrupts are an important synchronization mechanism in a real-time operating system because there will be multiple tasks to perform. To achieve real-time response interrupt-based synchronization serves as an important tool.

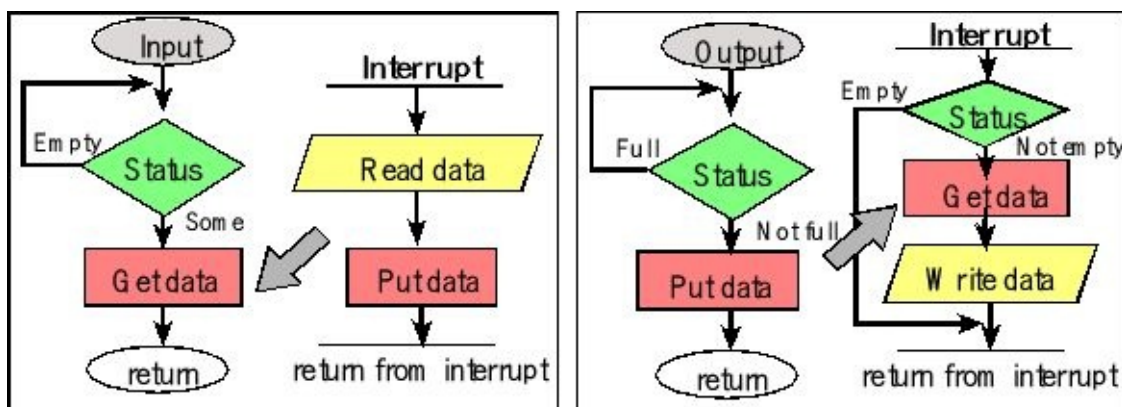


Figure 2.1. Flowcharts illustrating the use of interrupts for input and for output.

## 2.2.1. NVIC

On the ARM Cortex-M processor, exceptions include resets, software interrupts and hardware interrupts. Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located. Vectors are stored in ROM at the beginning of memory. Program 2.3 shows the first few vectors as defined in the `startup_TM4C123.s` file for the TM4C123 and the `startup_msp432.s` file for the MSP432. **DCD** is an assembler pseudo-op that defines a 32-bit constant. ROM location 0x0000.0000 has the initial stack pointer, and location 0x0000.0004 contains the initial program counter, which is called the **reset vector**. It holds the address of a function called the reset handler, which is the first thing executed following reset. There are hundreds of possible interrupt sources and their 32-bit vectors are listed in order starting with location 0x0000.0008. From a programming perspective, we can attach ISRs to interrupts by writing the ISRs as regular assembly subroutines or C functions with no input or output parameters and editing the `startup_TM4C123.s` or `startup_msp432.s` file to specify those functions for the appropriate interrupt. In this class, we will write our ISRs using standard function names so that the startup files need not be edited. For example, we will simply name the ISR for SysTick periodic interrupt as **SysTick\_Handler**. The ISR for this interrupt is a 32-bit pointer located at ROM address 0x0000.003C. Because the vectors are in ROM, this linkage is defined at compile time and not at run time. After the first 16 vectors, each processor will be different so check the data sheet.

```
EXPORT __Vectors
__Vectors                ; address  ISR
DCD StackMem + Stack    ; 0x00000000 Top of Stack
DCD Reset_Handler      ; 0x00000004 Reset Handler
DCD NMI_Handler        ; 0x00000008 NMI Handler
DCD HardFault_Handler  ; 0x0000000C Hard Fault Handler
DCD MemManage_Handler  ; 0x00000010 MPU Fault Handler
DCD BusFault_Handler   ; 0x00000014 Bus Fault Handler
DCD UsageFault_Handler ; 0x00000018 Usage Fault Handler
DCD 0                   ; 0x0000001C Reserved
DCD 0                   ; 0x00000020 Reserved
DCD 0                   ; 0x00000024 Reserved
DCD 0                   ; 0x00000028 Reserved
DCD SVC_Handler        ; 0x0000002C SVCcall Handler
DCD DebugMon_Handler   ; 0x00000030 Debug Monitor Handler
DCD 0                   ; 0x00000034 Reserved
DCD PendSV_Handler     ; 0x00000038 PendSV Handler
DCD SysTick_Handler    ; 0x0000003C SysTick Handler
```

*Program 2.3. Software syntax to set the interrupt vectors for the first 16 vectors on the Cortex M processor.*

Table 2.6 lists the interrupt sources we will use on the TM4C123 and Table 2.7 shows similar interrupts on the MSP432. Interrupt numbers 0 to 15 contain the faults, software interrupts and SysTick; these interrupts will be handled differently from interrupts 16 to 154.

Vector address	Number	IRQ	ISR name in <b>Startup.s</b>	NVIC priority	Priority bits
0x00000038	14	-2	<b>PendSV_Handler</b>	<b>SYS_PRI3</b>	23 – 21
0x0000003C	15	-1	<b>SysTick_Handler</b>	<b>SYS_PRI3</b>	31 – 29
0x000001E0	120	104	<b>WideTimer5A_Handler</b>	<b>NVIC_PRI26_R</b>	7 – 5

**Table 2.6. Some of the interrupt vectors for the TM4C (goes to number 154 on the M4).**

Vector address	Number	IRQ	ISR name in <b>Startup.s</b>	NVIC priority	Priority bits
0x00000038	14	-2	<b>PendSV_Handler</b>	<b>SYS_PRI3</b>	23 – 21
0x0000003C	15	-1	<b>SysTick_Handler</b>	<b>SYS_PRI3</b>	31 – 29
0x000000A4	41	25	<b>T32_INT1_IRQHandler</b>	<b>NVIC_IPR6</b>	15 – 13

**Table 2.7. Some of the interrupt vectors for the MSP432 (goes to number 154 on the M4).**

Interrupts on the Cortex-M are controlled by the Nested Vectored Interrupt Controller (NVIC). To activate an interrupt source we need to set its priority and enable that source in the NVIC. SysTick interrupt only requires arming the SysTick module for interrupts and enabling interrupts on the processor (I=0 in the **PRIMASK**). Other interrupts require additional initialization. In addition to arming and enabling, we will set bit 8 in the **NVIC\_EN3\_R** to activate **WideTimer5A** interrupts on the TM4C123. Similarly, we will set bit 25 in the **NVIC\_ISER0** to activate **T32\_INT1** interrupts on the MSP432. This activation is in addition to the arm and enable steps.

Each interrupt source has an 8-bit priority field. However, on the TM4C123 and MSP432 microcontrollers, only the top three bits of the 8-bit field are used. This allows us to specify the interrupt priority level for each device from 0 to 7, with 0 being the highest priority. The priority of the SysTick interrupt is found in bits 31 – 29 of the **SYS\_PRI3** register. Other interrupts have corresponding priority registers. The interrupt number (number column in Tables 2.6 and 2.7) is loaded into the **IPSR** register when an interrupt is being serviced. The servicing of interrupts does not set the I bit in the **PRIMASK**, so a higher priority interrupt can suspend the execution of a lower priority ISR. If a request of equal or lower priority is generated while an ISR is being executed, that request is postponed until the ISR is completed. In particular, those devices that need prompt service should be given high priority.

Figure 2.2 shows the context switch from executing in the foreground to running a SysTick periodic interrupt. The I bit in the **PRIMASK** is 0 signifying interrupts are enabled. Initially, the interrupt number (ISRNUM) in the **IPSR** register is 0, meaning we are running in **Thread mode** (i.e., the main program, and not an ISR). **Handler mode** is signified by a nonzero value in **IPSR**. When **BASEPRI** register is zero, all interrupts are allowed and the **BASEPRI** register is not active.



When a SysTick interrupt is triggered, the current instruction is finished. (a) Eight registers are pushed on the stack with **R0** on top. These registers are pushed onto the stack using whichever stack pointer is active: either the **MSP** or **PSP**. (b) The vector address is loaded into the **PC** (“Vector address” column in Tables 2.6 and 2.7). (c) The **IPSR** register is set to 15 (“Number” column in Tables 2.6 and 2.7) (d) The top 24 bits of **LR** are set to 0xFFFFFFFF, signifying the processor is executing an ISR. The bottom eight bits specify how to return from interrupt.

- 0xE1 Return to Handler mode MSP (using floating point state)
- 0xE9 Return to Thread mode MSP (using floating point state)
- 0xED Return to Thread mode PSP (using floating point state)
- 0xF1 Return to Handler mode MSP
- 0xF9 Return to Thread mode MSP ← we will mostly be using this one
- 0xFD Return to Thread mode PSP

After pushing the registers, the processor always uses the main stack pointer (**MSP**) during the execution of the ISR. Events b, c, and d can occur simultaneously.

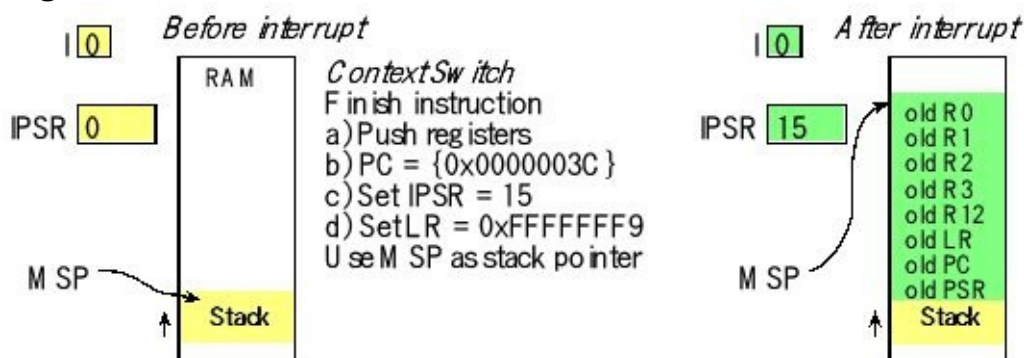


Figure 2.2. Stack before and after an interrupt, in this case a SysTick periodic interrupt.

To **return from an interrupt**, the ISR executes the typical function return statement: **BX LR**. However, since the top 24 bits of **LR** are 0xFFFFFFFF, it knows to return from interrupt by popping the eight registers off the stack. Since the bottom eight bits of **LR** in this case are 0b11111001, it returns to thread mode using the **MSP** as its stack pointer. Since the **IPSR** is part of the **PSR** that is popped, it is automatically reset to its previous state.

A **nested interrupt** occurs when a higher priority interrupt suspends an ISR. The lower priority interrupt will finish after the higher priority ISR completes. When one interrupt preempts another, the **LR** is set to 0xFFFFFFFF1, so it knows to return to handler mode. **Tail chaining** occurs when one ISR executes immediately after another. Optimization occurs because the eight registers need not be popped only to be pushed once again. If an interrupt is triggered and is in the process of stacking registers when a higher priority interrupt is requested, this **late arrival interrupt** will be executed first.

On the Cortex-M4, if an interrupt occurs while in the floating point state, an additional 18 words are pushed on the stack. These 18 words will save the state of the floating point processor. Bits 7-4 of the LR will be 0b1110 (0xE), signifying it was interrupted during a floating point state. When the ISR returns, it knows to pull these 18 words off the stack and restore the state of the floating point processor. We will not use floating point in this class.

**Priority** determines the order of service when two or more requests are made simultaneously. Priority also allows a higher priority request to suspend a lower priority request currently being processed. Usually, if two requests have the same priority, we do not allow them to interrupt each other. NVIC assigns a priority level to each interrupt trigger. This mechanism allows a higher priority trigger to interrupt the ISR of a lower priority request. Conversely, if a lower priority request occurs while running an ISR of a higher priority trigger, it will be postponed until the higher priority service is complete.

Program 2.4 shows two functions that can be used to enable and disable interrupts.

**DisableInterrupts**

**CPSID I**

**BX LR**

**EnableInterrupts**

**CPSIE I**

**BX LR**

*Program 2.4. Assembly functions needed for interrupt enabling and disabling.*

## 2.2.2. SysTick periodic interrupts

The SysTick Timer is a core device on the Cortex M architecture, which is most commonly used as a periodic timer. When used as a periodic timer one can setup the countdown to zero event to cause an interrupt. By setting up an initial reload value the timer is made to periodically interrupt at a predetermined rate decided by the reload value. Periodic timers as an interfacing technique are required for data acquisition and control systems, because software servicing must be performed at accurate time intervals. For a data acquisition system, it is important to establish an accurate sampling rate. The time in between ADC samples must be equal (and known) in order for the digital signal processing to function properly. Similarly, for microcontroller-based control systems, it is important to maintain both the input rate of the sensors and the output rate of the actuators. Periodic events are so important that most microcontrollers have multiple ways to generate periodic interrupts. **In this book our operating system will use periodic interrupts to schedule threads.**

Assume we have a 1-ms periodic interrupt. This means the interrupt service routine (ISR) is triggered to run 1000 times per second. Let **Count** be a global variable that

is incremented inside the ISR. Figure 2.3 shows how to use the interrupt to run Task 1 every  $N$  ms and run Task 2 every  $M$  ms.

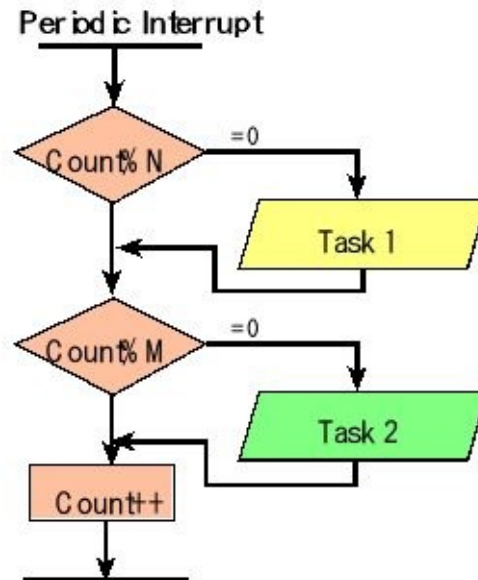


Figure 2.3. Using a 1-ms periodic interrupt to run Task 1 every  $N$  ms and run Task 2 every  $M$  ms.

The SysTick timer exists on all Cortex-M microcontrollers, so using SysTick means the system will be easy to port to other microcontrollers. Table 2.8 shows the register definitions for SysTick. The basis of SysTick is a 24-bit down counter that runs at the bus clock frequency. To configure SysTick for periodic interrupts we first clear the **ENABLE** bit to turn off SysTick during initialization, see Program 2.5. Second, we set the **STRELOAD** register. Third, we write any value to the **STCURRENT**, which will clear the counter and the flag. Lastly, we write the desired clock mode to the control register **STCTRL**, also setting the **INTEN** bit to enable interrupts and enabling the timer (**ENABLE**). We establish the priority of the SysTick interrupts using the **TICK** field in the **SYSPRI3** register. When the **STCURRENT** value counts down from 1 to 0, the **COUNT** flag is set. On the next clock, the **STCURRENT** is loaded with the **STRELOAD** value. In this way, the SysTick counter (**STCURRENT**) is continuously decrementing. If the **STRELOAD** value is  $n$ , then the SysTick counter operates at modulo  $n+1$ :

... $n, n-1, n-2 \dots 1, 0, n, n-1, \dots$

In other words, it rolls over every  $n+1$  counts. Thus, the **COUNT** flag will be configured to trigger an interrupt every  $n+1$  counts. The main program will enable interrupts in the processor after all variables and devices are initialized.

Address	31-24	23-17	16	15-3	2	1	0	Name
0xE000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	STCTRL
0xE000E014	0	24-bit RELOAD value						STRELOAD
0xE000E018	0	24-bit CURRENT value of SysTick counter						STCURRENT

--	--	--	--	--	--	--	--

Address	31-29	28-24	23-21	20-8	7-5	4-0	Name
0xE00ED20	TICK	0	PENDSV	0	DEBUG	0	SYSPRI3

**Table 2.8. SysTick registers.**

The SysTick counter decrements every bus cycle. So it is important to know the bus frequency when using SysTick. TM4C123 projects run at 16 MHz until the system calls a PLL function to change the frequency. MSP432 projects run at 3 MHz until the system calls a clock function to change the frequency. We will assume the MSP432 has been configured to run at its fastest speed of 48 MHz. In general, if the period of the core bus clock is  $t$  time units, then the **COUNT** flag will be set every  $(n+1)t$  time units. Reading the **STCTRL** control register will return the **COUNT** flag in bit 16, and then clear the flag. Also, writing any value to the **STCURRENT** register will reset the counter to zero and clear the **COUNT** flag. The **COUNT** flag is also cleared automatically as the interrupt service routine is executed.

Let  $f_{BUS}$  be the frequency of the bus clock, and let  $n$  be the value of the **STRELOAD** register. The frequency of the periodic interrupt will be

$$f_{BUS}/(n+1)$$

```
#define Profile_Toggle PC5^=0x20
void SysTick_Init(uint32_t period){
    Profile_Init(); // make PC5 is an output
    Counts = 0;
    STCTRL = 0; // disable SysTick during setup
    STRELOAD = period-1;// reload value
    STCURRENT = 0; // any write to current clears it
    SYSPRI3 = (SYSPRI3&0x00FFFFFF)|0x40000000; // priority 2
    STCTRL = 0x07; // enable, core clock, interrupts
}
void SysTick_Handler(void){ // Executed every (bus cycle)*(period)
    Profile_Toggle(); // toggle bit
    Profile_Toggle(); // toggle bit
    Counts = Counts + 1;
    Profile_Toggle(); // toggle bit
}
int main(void){ // TM4C123 bus clock at 16 MHz
    SysTick_Init(1600000); // SysTick timer interrupts every 100 ms
    EnableInterrupts();
    while(1){
        } // do nothing foreground
}
```

*Program 2.5. Implementation of a periodic interrupt using SysTick (SysTickInts\_xxx).*

**Checkpoint 2.2:** If the MSP432 bus clock is 48 MHz, what reload value yields a 100 Hz (10ms) periodic interrupt?

### 2.2.3. Periodic timer interrupts

Because time is a precious commodity for embedded systems there is a rich set of features available to manage time. If you connect a digital input to the microcontroller you could measure its

Period, time from one edge to the next

Frequency, number of edges in a fixed amount of time

Pulse width, time the signal is high, or time the signal is low

If there are multiple digital inputs, then you can measure more complicated parameters such as frequency difference, period difference or phase.

Alternately, you can create a digital output and have the software set its

Period

Frequency

Duty cycle (pulse-width modulation)

If there are multiple digital outputs, then you can create more complicated patterns that are used in stepper motor and brushless DC motor controllers. For examples of projects that manage time on the TM4C123 see examples at

<http://users.ece.utexas.edu/~valvano/arm/#Timer>

<http://edx-org-utaustinx.s3.amazonaws.com/UT601x/ValvanoWareTM4C123.zip>

For all the example projects on the TM4C123/MSP432 download and unzip these projects:

<http://edx-org-utaustinx.s3.amazonaws.com/UT601x/ValvanoWare.zip>

However, in this section, we present the basic principles needed to create periodic interrupts using the timer. We begin by presenting five hardware components needed as shown in Figure 2.4.

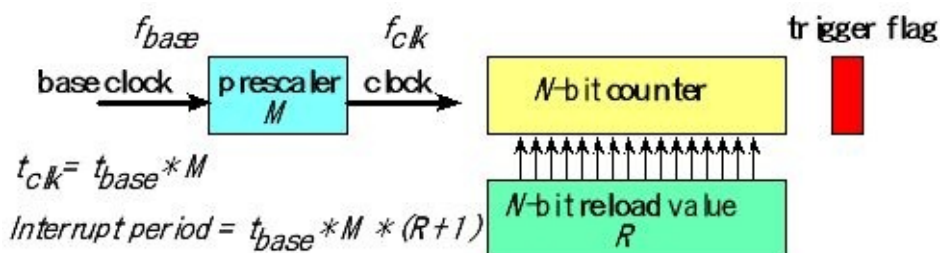


Figure 2.4. Fundamental hardware components used to create periodic

## *interrupts.*

The central component for creating periodic interrupts is a **hardware counter**. The counter may be 16, 24, 32, 48, or 64 bits wide. Let  $N$  be the number of bits in the counter. When creating periodic interrupts, it doesn't actually matter if the module counts up or counts down. However, most of the software used in this class will configure the counter to decrement.

Just like SysTick, as the counter counts down to 0, it sets a trigger flag and reloads the counter with a new value. The second component will be the **reload value**, which is the  $N$ -bit value loaded into the counter when it rolls over. Typically, the reload value is a constant set once by the software during initialization. Let  $R$  be this constant value.

The third component is the **trigger flag**, which is set when the counter reaches 0. This flag will be armed to request an interrupt. Software in the ISR will execute code to acknowledge or clear this flag.

The fourth component will be the **base clock** with which we control the entire hardware system. On the TM4C123, we will select the 80-MHz system clock. On the MSP432, we will select the 12-MHz SMCLK. In both cases, these clocks are derived from the crystal; hence timing will be both accurate and stable. Let  $f_{base}$  be the frequency of the base clock (80 MHz or 12 MHz) and  $t_{base}$  be the period of this clock (12.5 ns or about 83.33 ns).

The fifth component will be a **prescaler**, which sits between the base clock and the clock used to decrement the counter. Most systems create the prescaler using a modulo- $M$  counter, where  $M$  is greater than or equal to 1. This way, the frequency and period of the clock used to decrement the counter will be

$$f_{clk} = f_{base} / M \qquad t_{clk} = t_{base} * M$$

Software can configure the prescaler to slow down the counting. However, the interrupt period will be an integer multiple of  $t_{clk}$ . In addition, the interrupt period must be less than  $2^N * t_{clk}$ . Thus, the smaller the prescale  $M$  is, the finer control the software has in selecting the interrupt period. On the other hand, the larger prescale  $M$  is, the longer the interrupt could be. Thus, the prescaler allows the software to control the tradeoff between maximum interrupt period and the fine-tuning selection of the interrupt period.

Because the counter goes from the reload value down to 0, and then back to the reload value, an interrupt will be triggered every  $R+1$  counts. Thus the interrupt period,  $P$ , will be

$$P = t_{base} * M * (R + 1)$$

Solving this equation for  $R$ , if we wish to create an interrupt with period  $P$ , we make

$$R = (P / (t_{base} * M)) - 1$$

Remember  $R$  must be an integer less than  $2^N$ . Most timers have a limited choice for the prescale  $M$ . Luckily, most microcontrollers have a larger number of timers. The TM4C123 has six 32-bit timers and six 64-bit timers. The MSP432 has four 16-bit timers and two 32-bit timers. The board support package, presented in the next section, provides support for two independent periodic interrupts. Initialization software follows these steps.

- 0) Activate the base clock for the timer
- 1) Disable timer during initialization
- 2) Set the timer mode to continuous down counting with automatic reload
- 3) Set the reload value,  $R$
- 4) Set the prescale,  $M$
- 5) Arm the trigger flag in the timer
- 6) Arm the timer in the NVIC
- 7) Set the priority in the NVIC
- 8) Clear trigger flag
- 9) Enable timer after timer is completely configured
- 10) Enable interrupts ( $I=0$ ), typically done after all initializations are complete

For more details on the timers for the TM4C123 or MSP432, see the corresponding Volume 2. However, we present one simple solution that executes a user task at a periodic rate with units of  $\mu\text{s}$ . We will generate a periodic interrupt and call the user task from the ISR. Assuming an 80 MHz bus clock, we disable the prescale, meaning the timer counts every 12.5ns. To define the user task, we will create a private global variable containing a pointer to the user's function. We will set the variable during initialization and call that function at run time. Another name for a dynamically set function pointer is a **hook**. The maximum possible value for **period** is  $12.5\text{ns} * 2^{32}$ , which is about 53 seconds. Simple solutions for the TM4C and MSP432 are shown in Program 2.6. You will find many more on the book web site.

```

void (*PeriodicTask)(void); // user function
void Timer0B_Init(void(*task)(void), uint32_t period){
    SYSCTL_RCGCTIMER_R |= 0x0001; // 0) activate timer0
    PeriodicTask = task; // user function
    TIMER0_CTL_R &= ~0x00000100; // 1) disable timer0B during setup
    TIMER0_CFG_R = 0x00000000; // 2) configure for 32-bit timer mode
    TIMER0_TBMR_R = 0x00000002; // configure for periodic mode
    TIMER0_TBILR_R = period-1; // 3) reload value
    TIMER0_TBPR_R = 0; // 4) no prescale, 12.5ns clock
    TIMER0_IMR_R |= 0x00000100; // 5) arm timeout interrupt
    NVIC_EN0_R = (1<<20); // 6) enable interrupt 20 in NVIC

```

```

NVIC_PRI5_R = (NVIC_PRI5_R&0xFFFFF00)|0x00000040; // 7) priority 2
TIMER0_ICR_R = 0x00000100;    // 8) clear timer0B timeout flag
TIMER0_CTL_R |= 0x00000100;    // 9) enable timer0B
}
void Timer0B_Handler(void){
    TIMER0_ICR_R = 0x00000100;    // acknowledge timer0B timeout
    (*PeriodicTask());           // execute user task
}

```

*Program 2.6a. Implementation of a periodic interrupt using Timer0B (see PeriodicTimer0AInts\_xxx).*

```

void TimerA0_Init(void(*task)(void), uint16_t period){
    PeriodicTask = task;          // user function
    TA0CTL &= ~0x0030;           // 1) halt Timer A0
    TA0CTL = 0x0202;             // 2) compare mode
    TA0CCTL0 = 0x0010;
    TA0CCR0 = (period - 1);      // 3) compare match value
    TA0EX0 &= ~0x0007;          // 4) input clock divider /1
    NVIC_ISER0 = 0x00000100;     // 6) enable interrupt 8 in NVIC
    NVIC_IPR2 = (NVIC_IPR2&0xFFFFF00)|0x00000040; // 7) priority 2
    TA0CCTL0 &= ~0x0001;        // 8) clear interrupt flag 0
    TA0CTL |= 0x0014;           // 5,9) reset and start Timer A0 in up mode
}
void TA0_0_IRQHandler(void){
    TA0CCTL0 &= ~0x0001;        // acknowledge capture/compare interrupt 0
    (*PeriodicTask());          // execute user task
}

```

*Program 2.6b. Implementation of a periodic interrupt using Timer0B (see PeriodicTimerA0Ints\_xxx).*

## 2.2.4. Critical sections

An important consequence of multi-threading is the potential for the threads to manipulate (read/write) a shared object. With this potential comes the possibility of inconsistent updates to the shared object. A **race condition** occurs in a multi-threaded environment when there is a causal or timing dependency between two or more threads. In other words, different behavior occurs depending on the order of execution of two threads. Consider a simple example of a race condition occurring where two thread initialize the same port in an unfriendly manner. Thread-1 initializes Port 4 bits 3 – 0 to be output using **P4DIR** = 0x0F; Thread-2 initializes Port 4 bits 6 – 4 to be output using **P4DIR** = 0x70; In particular, if Thread-1 runs first and Thread-2 runs second, then Port 4 bits 3 – 0 will be set to inputs. Conversely, if



Thread-2 runs first and Thread-1 runs second, then Port 4 bits 6 – 4 will be set to inputs. This is a race condition caused by unfriendly code. The solution to this problem is to write the two initializations in a friendly manner, and make both initializations atomic.

In a second example of a race condition, assume two threads are trying to get data from the same input device. Both call the input function to receive data from the input device. When data arrives at the input, the thread that executes first will capture the data.

In general, if two threads access the same global memory and one of the accesses is a write, then there is a **causal dependency** between the execution of the threads. Such dependencies when not properly handled cause unpredictable behavior where the execution order may affect the outcome. Such scenarios are referred to as **race conditions**. While shared global variables are important in multithreaded systems because they are required to pass data between threads, they result in complex behavior (and hard to find bugs). Therefore, a programmer must pay careful attention to avoid race conditions.

A program segment is **reentrant** if it can be concurrently executed by two (or more) threads. Note that, to run concurrently means both threads are ready to run though only one thread is currently running. To implement reentrant software, we place variables in registers or on the stack, and avoid storing into global memory variables. When writing in assembly, we use registers, or the stack for parameter passing to create reentrant subroutines. Typically, each thread will have its own set of registers and stack. A non-reentrant subroutine will have a section of code called a **vulnerable window** or **critical section**. A critical section may exist when two different functions access and modify the same memory-resident data structure. E.g.,

- 1) One thread calls a non-reentrant function
- 2) It is executing in the critical section when interrupted by a second thread
- 3) The second thread calls the same non-reentrant function.

There are a number of scenarios that can happen next. In the most common scenario, the second thread is allowed to complete the execution of the function, control is then returned to the first thread, and the first thread finishes the function. This first scenario is the usual case with interrupt programming. In the second scenario, the second thread executes part of the critical section, is interrupted and then re-entered by a third thread, the third thread finishes, the control is returned to the second thread and it finishes, lastly the control is returned to the first thread and it finishes. This second scenario can happen in interrupt programming if the second interrupt has higher priority than the first.

Program 2.7 shows two C functions and the corresponding assembly codes. These functions have critical sections because of their read-modify-write nonatomic access to the global variable, **count**. If an interrupt were to occur just before or just after the **ADD** or **SUB** instruction, and the ISR called the other function, then count would

be in error.

<pre>count SPACE 4 ← Producer LDR r1,[pc,#116] ; R0= &amp;count ←          LDR r0,[r1]      ; R0=count     ADD r0,r0,#1     STR r0,[r1] ; update     BX lr ← ← Consumer LDR r1, [pc,#96] ; R0= &amp;count     LDR r0,[r1] ; R0=count     SUB r0,r0,#1     STR r0,[r1] ; update     BX lr DCD num</pre>	<pre>int32_t volatile count; void Producer(void){     // other stuff     count = count + 1;     // other stuff } void Consumer(void){     // other stuff     count = count - 1;     // other stuff }</pre>
--	--

*Program 2.7. These functions are nonreentrant because of the read-modify-write access to a global. The critical section, pointed to by arrows, is just before and just after the ADD and SUB instructions.*

Assume there are two concurrent threads, where the main program calls **Producer** and a background ISR calls **Consumer**. Concurrent means that both threads are ready to run. Because there is only one computer, exactly one thread will be running at a time. Typically, the operating system switches execution control back and forth using interrupts. There are two places in the assembly code of **Producer** at which if an interrupt were to occur and the ISR called the **Consumer** function, the end value of count will be inconsistent. Assume for this example **count** is initially 4. An error occurs if:

1. The main program calls **Producer**
2. The main executes **LDR r0,[r1]** making R0 = 4
3. The OS suspends the main (using an interrupt) and starts the ISR
4. The ISR calls **Consumer**  
Executes **count=count-1**; making **count** equal to 3
5. The OS returns control back to the main program  
R0 is back to its original value of 4
6. The producer finishes (adding 1 to R0)  
Making **count** equal to 5

The expected behavior with the producer and consumer executing once is that count would remain at 4. However, the race condition resulted in an inconsistency manifesting as a lost consumption. As the reader may have observed, the cause of the problem is the non-atomicity of the read-modify-write operation involved in reading and writing to the count ( `count=count+1` or `count=count-1` ) variable. An **atomic operation** is one that once started is guaranteed to finish. In most computers, once an assembly instruction has begun, the instruction must be finished before the computer can process an interrupt. The same is not the case with C instructions which themselves translate to multiple assembly instructions. In general, nonreentrant code can be grouped into three categories all involving 1) nonatomic sequences, 2) writes and 3) global variables. We will classify I/O ports as global variables for the consideration of critical sections. We will group registers into the same category as local variables because each thread will have its own registers and stack.

The first group is the **read-modify-write** sequence:

1. The software reads the global variable producing a copy of the data
2. The software modifies the copy (original variable is still unmodified)
3. The software writes the modification back into the global variable.

In the second group, we have a **write followed by read**, where the global variable is used for temporary storage:

1. The software writes to the global variable (only copy of the information)
2. The software reads from the global variable expecting the original data to be there.

In the third group, we have a **non-atomic multi-step write** to a global variable:

1. The software writes part of the new value to a global variable
2. The software writes the rest of the new value to a global variable.

**Observation:** When considering reentrant software and vulnerable windows we classify accesses to I/O ports the same as accesses to global variables.

**Observation:** Sometimes we store temporary information in global variables out of laziness. This practice is to be discouraged because it wastes memory and may cause the module to not be reentrant.

Sometimes we can have a critical section between two different software functions (one function called by one thread, and another function called by a different thread).

In addition to above three cases, a **non-atomic multi-step read** will be critical when paired with a **multi-step write**. For example, assume a data structure has multiple components (e.g., hours, minutes, and seconds). In this case, the write to the data structure will be atomic because it occurs in a high priority ISR. The critical section exists in the foreground between steps 1 and 3. In this case, a critical section exists even though no software has actually been reentered.

<u>Foreground thread</u>	<u>Background thread</u>
1. The main reads some of the data	2. ISR writes to the data structure
3. The main reads the rest of the data	

In a similar case, a **non-atomic multi-step write** will be critical when paired with a **multi-step read**. Again, assume a data structure has multiple components. In this case, the read from the data structure will be atomic because it occurs in a high priority ISR. The critical section exists in the foreground between steps 1 and 3.

<u>Foreground thread</u>	<u>Background thread</u>
1. The main writes some of the data	2. ISR reads from the data structure
3. The main writes the rest of the data	

When multiple threads are active, it is possible for two threads to be executing the same program. For example, the system may be running in the foreground and calls a function. Part way through execution of the function, an interrupt occurs. If the ISR also calls the same function, two threads are simultaneously executing the function.

If critical sections do exist, we can either eliminate them by removing the access to the global variable or implement **mutual exclusion**, which simply means only one thread at a time is allowed to execute in the critical section. In general, if we can eliminate the global variables, then the subroutine becomes reentrant. Without global variables there are no “vulnerable” windows because each thread has its own registers and stack. Sometimes one must access global memory to implement the desired function. Remember that all I/O ports are considered global. Furthermore, global variables are necessary to pass data between threads. Program 2.8 shows two functions that can be used to implement mutual exclusion.

```

;***** StartCritical *****
; make a copy of previous I bit, disable interrupts
; inputs: none      voutputs: previous I bit
StartCritical
    MRS  R0, PRIMASK ; save old status
    CPSID I          ; mask all (except faults)

```

```

    BX    LR
;***** EndCritical *****
; using the copy of previous I bit, restore I bit to previous value
; inputs: previous I bit  outputs: none
EndCritical
    MSR  PRIMASK, R0
    BX    LR

```

*Program 2.8. Assembly functions needed to implement mutual exclusion.*

A simple way to implement mutual exclusion is to disable interrupts while executing the critical section. It is important to disable interrupts for as short a time as possible, so as to minimize the effect on the dynamic performance of the other threads. While we are running with interrupts disabled, time-critical events like power failure and danger warnings cannot be processed. The assembly code of Program 2.8 is in the startup file in our projects that use interrupts. Program 2.9 illustrates how to implement mutual exclusion and eliminate the critical section.

When making code atomic with this simple method, make sure one critical section is not nested inside another critical section.

```

    uint32_t volatile count; // number of elements
    // simple option

```

<pre> void Producer(void){     DisableInterrupts();     count = count + 1;     EnableInterrupts(); } </pre>	<pre> void Consumer(void){     DisableInterrupts();     count = count - 1;     EnableInterrupts(); } </pre>
---	---

```

    // safer option

```

<pre> void Producer(void){     long sr;     sr = StartCritical();     count = count + 1;     EndCritical(sr); } </pre>	<pre> void Consumer(void){     long sr;     sr = StartCritical();     count = count - 1;     EndCritical(sr); } </pre>
--	--

*Program 2.9. These functions are reentrant because of the read-modify-write access to the global is atomic. Use the simple option only if one critical section is not nested inside another critical section.*

**Checkpoint 2.3:** Although disabling interrupts does remove critical sections, it will add latency and jitter to real-time systems. Explain how latency and jitter are affected by the `DisableInterrupts()` and `EnableInterrupts()` functions.

**Checkpoint 2.4:** Consider the situation of nested critical sections. For example, a

function with a critical section calls another function that also has a critical section. What would happen if you simply added disable interrupts at the beginning and a re-enable interrupts at the end of each critical section?

## 2.2.5. Executing periodic tasks

The timers provide a simple way to execute periodic tasks. A periodic task is one that is performed on a fixed time basis. This interfacing technique is required for data acquisition and control systems, because software servicing must be performed at accurate time intervals. For a data acquisition system, it is important to establish an accurate sampling rate. The time in between ADC samples must be equal (and known) in order for the digital signal processing to function properly. Similarly, for microcontroller-based control systems, it is important to maintain both the ADC and DAC timing. The general purpose timers can also create periodic interrupts. The operating system will use periodic interrupts to schedule threads.

Another application of periodic interrupts is called “intermittent polling” or “periodic polling”. Figure 2.5 shows busy wait side by side with periodic polling. In busy-wait synchronization, the main program polls the I/O devices continuously. With periodic polling, the I/O devices are polled on a regular basis (established by the periodic interrupt.) If no device needs service, then the interrupt simply returns. If the polling period is  $\Delta t$ , then on average the interface latency will be  $\frac{1}{2}\Delta t$ , and the worst case latency will be  $\Delta t$ . Periodic polling is appropriate for low bandwidth devices where real-time response is not necessary. This method frees the main program from the I/O tasks.

We use periodic polling if the following two conditions apply:

1. The I/O hardware cannot generate interrupts directly
2. We wish to perform the I/O functions in the background

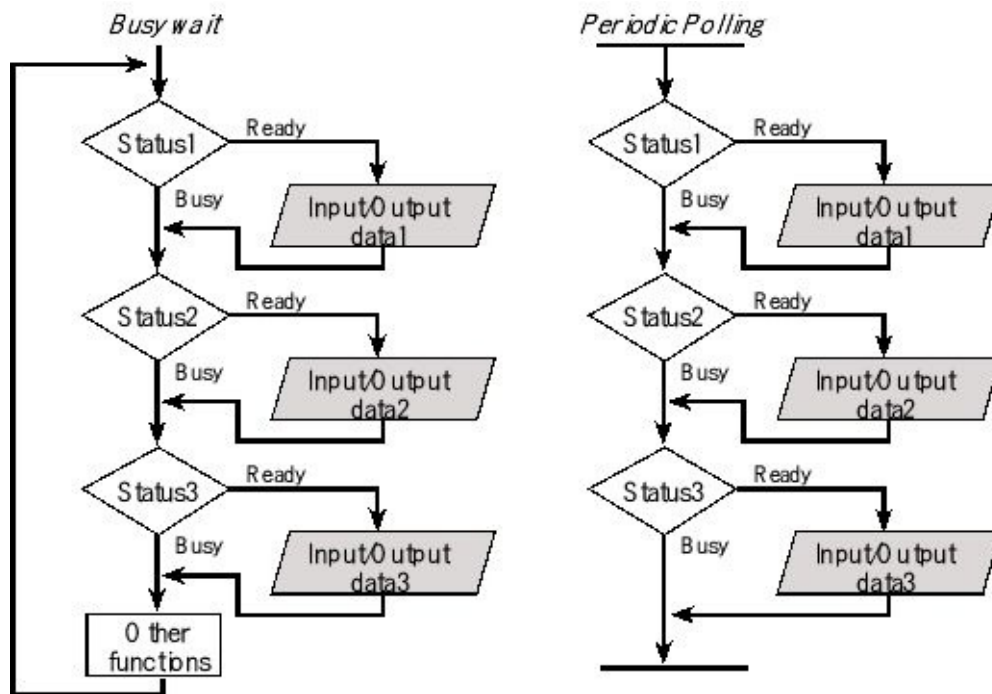


Figure 2.5. An ISR flowchart that implements periodic polling.

## 2.2.6. Software interrupts

When the user code is not compiled and linked together with the operating system, the user code can invoke the OS using the supervisor call instruction, **SVC**. A software interrupt, or trap, is a software-triggered interrupt. In the user code, various OS functions can be invoked with specifying a trap number to the **SVC** instruction

**OS\_Sleep**  
**SVC #2**  
**BX LR**

**OS\_Time**  
**SVC #3**  
**BX LR**

On the Cortex M, the **SVC** instruction will invoke a software interrupt, which is similar to hardware interrupts in that 8 registers are pushed on the stack and the PC is loaded with the corresponding ISR vector address. Within the OS, the **SVC** handler will look into the object code of the **SVC** instruction to extract the trap number, which will be the least significant 8 bits of the 16-bit instruction. If the OS function has input or output parameters they will be passed and returned on the stack, rather than in registers.

**SVC\_Handler**  
**LDR R12,[SP,#24] ; Return address**  
**LDRH R12,[R12,#-2] ; SVC instruction is 2 bytes**

```
BIC R12,#0xFF00 ; Extract trap number in R12  
LDM SP,{R0-R3} ; Get any parameters  
...  
BL OS_xxx ; Call OS routine by number  
...  
STR R0,[SP] ; Store return value  
BX LR ; Return from exception
```

**PendSV** is similar to SVC in that the interrupt is invoked by software and not hardware. To trigger a PendSV interrupt we write a 1 to bit 28 of the interrupt control register. PendSV does not have a trap number, so we typically use it for just one dedicated purpose.

```
INTCTRL = 0x10000000; // trigger PendSV
```

Similarly, software can trigger a SysTick interrupt by writing a 1 to bit 26.

```
INTCTRL = 0x04000000; // trigger SysTick
```



---

## 2.3. First in First Out (FIFO) Queues

The first in first out (FIFO) queue is an important data structure for I/O programming because it allows us to pass data from one module to another. One module puts data into the FIFO and another module gets data out of the FIFO. Programs 2.10 and 2.11 define macros allowing us to create as many FIFOs as we need. These FIFO implementations are meant for embedded systems without an operating system, hence they do not include semaphore synchronization.

```
// macro to create a pointer FIFO
#define AddPointerFifo(NAME,SIZE,TYPE,SUCCESS,FAIL) \
TYPE volatile *NAME ## PutPt; \
TYPE volatile *NAME ## GetPt; \
TYPE static NAME ## Fifo [SIZE]; \
void NAME ## Fifo_Init(void){ \
    NAME ## PutPt = NAME ## GetPt = &NAME ## Fifo[0]; \
} \
int NAME ## Fifo_Put (TYPE data){ \
    TYPE volatile *nextPutPt; \
    nextPutPt = NAME ## PutPt + 1; \
    if(nextPutPt == &NAME ## Fifo[SIZE]){ \
        nextPutPt = &NAME ## Fifo[0]; \
    } \
    if(nextPutPt == NAME ## GetPt ){ \
        return(FAIL); \
    } \
    else{ \
        *( NAME ## PutPt ) = data; \
        NAME ## PutPt = nextPutPt; \
        return(SUCCESS); \
    } \
} \
int NAME ## Fifo_Get (TYPE *datap){ \
    if( NAME ## PutPt == NAME ## GetPt ){ \
        return(FAIL); \
    } \
    *datap = *( NAME ## GetPt ## ++); \
    if( NAME ## GetPt == &NAME ## Fifo[SIZE]){ \
        NAME ## GetPt = &NAME ## Fifo[0]; \
    } \
    return(SUCCESS); \
}
```

*Program 2.10. Two-pointer macro implementation of a FIFO.*

To create a 20-element FIFO storing unsigned 16-bit numbers that returns 1 on success and 0 on failure we invoke

**AddPointerFifo(Rx, 20, uint16\_t, 1, 0)**

creating the three functions **RxFifo\_Init()** , **RxFifo\_Get()** ,and **RxFifo\_Put()** .

Program 2.11 is a macro we can use to create two-index FIFOs.

```
// macro to create an index FIFO
#define AddIndexFifo(NAME,SIZE,TYPE,SUCCESS,FAIL) \
uint32_t volatile NAME ## PutI; \
uint32_t volatile NAME ## GetI; \
TYPE static NAME ## Fifo [SIZE]; \
void NAME ## Fifo_Init(void){ \
    NAME ## PutI = NAME ## GetI = 0; \
} \
int NAME ## Fifo_Put (TYPE data){ \
    if(( NAME ## PutI - NAME ## GetI ) & ~(SIZE-1)){ \
        return(FAIL); \
    } \
    NAME ## Fifo[ NAME ## PutI &(SIZE-1)] = data; \
    NAME ## PutI ## ++; \
    return(SUCCESS); \
} \
int NAME ## Fifo_Get (TYPE *datap){ \
    if( NAME ## PutI == NAME ## GetI){ \
        return(FAIL); \
    } \
    *datap = NAME ## Fifo[ NAME ## GetI &(SIZE-1)]; \
    NAME ## GetI ## ++; \
    return(SUCCESS); \
} \
uint16_t NAME ## Fifo_Size (void){ \
return ((uint16_t)( NAME ## PutI - NAME ## GetI )); \
}
```

*Program 2.11. Macro implementation of a two-index FIFO. The size must be a power of two.*

To create a 32-element FIFO storing signed 32-bit numbers that returns 0 on success and 1 on failure we invoke

**AddIndexFifo(Tx, 32, int32\_t, 0, 1)**

creating the four functions **TxFifo\_Init()**, **TxFifo\_Get()**, **TxFifo\_Put()**, and **TxFifo\_Size()**. We can use the following macro to collect histogram data. Basically, we can add **Collect()** to places where data are added to the FIFO.

```
#define Collect() (Histogram[TxFifo_Size()]++)
```



\$4000.4518	SLR	SLR	SLR	SLR	SLR	SLR	SLR	SLR	GPIO_PORTA_SLR_R
\$4000.451C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTA_DEN_R
\$4000.4524	CR	CR	CR	CR	CR	CR	CR	CR	GPIO_PORTA_CR_R
\$4000.4528	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	AMSEL	GPIO_PORTA_AMSEL_R
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0	
\$4000.452C	PMC7	PMC6	PMC5	PMC4	PMC3	PMC2	PMC1	PMC0	GPIO_PORTA_PCTL_R
\$4000.4520	LOCK (write 0x4C4F434B to unlock, other locks) (reads 1 if locked, 0 if unlocked)								GPIO_PORTA_LOCK_R

**Table 2.9. Port A registers for the TM4C.**

We clear bits in the **AMSEL** register to use the port for digital I/O. **AMSEL** bits exist for those pins which have analog functionality. We set the alternative function using both **AFSEL** and **PCTL** registers. On the TM4C123, we need to unlock PD7 and PF0 if we wish to use them. On the TM4C1294, only PD7 needs unlocking. Because PC3-0 implements the JTAG debugger, we will never unlock these pins. To unlock a pin, we first write 0x4C4F434B to the **LOCK** register, and then we write zeros to the **CR** register.

To configure an edge-triggered pin, we first enable the clock on the port and configure the pin as a regular digital input. We can trigger on the rising, falling, or both edges, as listed in Table 2.10. Clearing the **IS** (Interrupt Sense) bit configures the bit for edge triggering. If the **IS** bit were to be set, the trigger occurs on the level of the pin.

DIR	AFSEL	IS	IBE	IEV	IME	Port mode
0	0	0	0	0	0	Input, falling edge trigger, busy wait
0	0	0	0	1	0	Input, rising edge trigger, busy wait
0	0	0	1	-	0	Input, both edges trigger, busy wait
0	0	0	0	0	1	Input, falling edge trigger, interrupt
0	0	0	0	1	1	Input, rising edge trigger, interrupt
0	0	0	1	-	1	Input, both edges trigger, interrupt

**Table 2.10. Edge-triggered modes.**

Since most busy to done conditions are signified by edges, we typically trigger on edges rather than levels. Next we write to the **IBE** (Interrupt Both Edges) and **IEV** (Interrupt Event) bits to define the active edge. We clear the **IME** (Interrupt Mask Enable) bits if we are using busy-wait synchronization, and we set the **IME** bits to use interrupt synchronization.

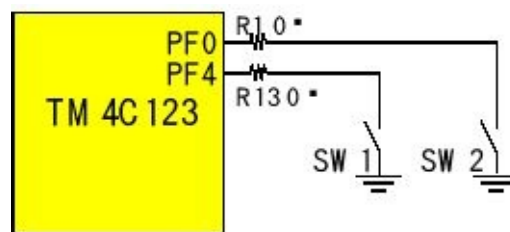
The hardware sets an **RIS** (Raw Interrupt Status) bit (called the trigger) and the software clears it (called the acknowledgement). The triggering event listed in Table

2.10 will set the corresponding **RIS** bit in the **GPIO\_PORTA\_RIS\_R** register regardless of whether or not that bit is allowed to request a controller interrupt. In other words, clearing an **IME** bit disables the corresponding pin's interrupt, but it will still set the corresponding **RIS** bit when the interrupt would have occurred. The software can acknowledge the event by writing ones to the corresponding **IC**(Interrupt Clear) bit in the **GPIO\_PORTA\_IC\_R** register. The **RIS** bits are read only, meaning if the software were to write to this registers, it would have no effect. For example, to clear bits 2, 1, and 0 in the **GPIO\_PORTA\_RIS\_R** register, we write a 0x07 to the **GPIO\_PORTA\_IC\_R** register. Writing zeros into **IC** bits will not affect the **RIS** bits.

For input signals we have the option of adding either a pull-up resistor or a pull-down resistor. If we set the corresponding **PUE** (Pull-Up Enable) bit on an input pin, the equivalent of a 50 to 110 kΩ resistor to +3.3 V power is internally connected to the pin. Similarly, if we set the corresponding **PDE** (Pull-Down Enable) bit on an input pin, the equivalent of a 55 to 180 kΩ resistor to ground is internally connected to the pin. We cannot have both pull-up and a pull-down resistor, so setting a bit in one register automatically clears the corresponding bit in the other register.

A typical application of pull-up and pull-down mode is the interface of simple switches. Using these modes eliminates the need for an external resistor when interfacing a switch. The switch interfaces for the two switches on the LaunchPad are illustrated in Figure 2.6. The Port F interfaces employ software-configured internal resistors, implementing negative logic inputs.

**Checkpoint 2.5:** What do negative logic and positive logic mean in the context of interfacing switches?



*Figure 2.6. Edge-triggered interfaces can generate interrupts on a switch touch. These negative logic switches require internal pull-up resistors. R1 and R13 are 0-ohm resistors can could be desoldered to disconnect the switches from the microcontroller.*

**Checkpoint 2.6:** What values to you write into DIR, AFSEL, PUE, and PDE to configure the switch interfaces of PF4 and PF0 in Figure 2.6?

Using edge triggering to synchronize software to hardware centers around the operation of the trigger flags, **RIS**. A busy-wait interface will read the appropriate **RIS** bit over and over, until it is set. When the **RIS** bit is set, the software will clear the **RIS** bit (by writing a one to the corresponding **IC** bit) and perform the desired function. With interrupt synchronization, the initialization phase will arm the trigger

flag by setting the corresponding **IME** bit. In this way, the active edge of the pin will set the **RIS** and request an interrupt. The interrupt will suspend the main program and run a special interrupt service routine (ISR). This ISR will clear the **RIS** bit and perform the desired function. At the end of the ISR it will return, causing the main program to resume. In particular, five conditions must be simultaneously true for an edge-triggered interrupt to be requested:

- The trigger flag bit is set (RIS)
- The arm bit is set (IME)
- The level of the edge-triggered interrupt must be less than **BASEPRI**
- The edge-triggered interrupt must be enabled in the **NVIC\_EN0\_R**
- The edge-triggered interrupt must be disabled in the **NVIC\_DIS0\_R**
- Bit 0 of the special register **PRIMASK** is 0

Table 2.9 listed the registers for Port A. The other ports have similar registers. We will begin with a simple example that counts the number of falling edges on Port F bits 4,0 (Program 2.12). The initialization requires many steps. (a) The clock for the port must be enabled. (b) The global variables should be initialized. (c) The appropriate pins must be enabled as inputs. (d) We must specify whether to trigger on the rise, the fall, or both edges. In this case, we will trigger on the fall of PF4,PF0. (e) It is good design to clear the trigger flag during initialization so that the first interrupt occurs due to the first rising edge after the initialization has been run. We do not wish to trigger on a falling edge that might have occurred during the power up phase of the system. (f) We arm the edge-trigger by setting the corresponding bits in the **IM**register. (g) We establish the priority of Port F by setting bits 23 – 21 in the **NVIC\_PRI7\_R** register. We activate Port F interrupts in the NVIC by writing a one to bit 30 in the **NVIC\_EN0\_R** register (“IRQ number”). In most systems we would not enable interrupts in the device initialization. Rather, it is good design to initialize all devices in the system, and then enable interrupts.

**Checkpoint 2.7:** If both switches are touched simultaneously, what will happen? How many interrupts are generated?

```
int32_t Count1,Count2 = 0;
void Switch_Init(void){
    SYSCTL_RCGCGPIO_R |= 0x20;    // (a) activate clock for Port F
    Count1= Count2 = 0;          // (b) initialize counters
    GPIO_PORTF_LOCK_R = 0x4C4F434B; // unlock GPIO Port F
    GPIO_PORTF_CR_R = 0x1F;      // allow changes to PF4-0
    GPIO_PORTF_DIR_R = 0x02;     // (c) make PF4,PF0 in and PF1 is out
    GPIO_PORTF_DEN_R |= 0x13;    // enable digital I/O on PF4,PF0, PF1
    GPIO_PORTF_PUR_R |= 0x11;    // pullups on PF4,PF0
```

```

GPIO_PORTF_IS_R &= ~0x11;    // (d) PF4,PF0 are edge-sensitive
GPIO_PORTF_IBE_R &= ~0x11;    // PF4,PF0 are not both edges
GPIO_PORTF_IEV_R &= ~0x11;    // PF4,PF0 falling edge event
GPIO_PORTF_ICR_R = 0x11;    // (e) clear flags
GPIO_PORTF_IM_R |= 0x11;    // (f) arm interrupt on PF4,PF0
NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00A00000; // (g) priority 5
NVIC_EN0_R = 0x40000000;    // (h) enable interrupt 30 in NVIC
}
void GPIOPortF_Handler(void){
    if(GPIO_PORTF_RIS_R&0x10){ // poll PF4
        GPIO_PORTF_ICR_R = 0x10; // acknowledge flag4
        Count1++;           // event occurred
    }
    if(GPIO_PORTF_RIS_R&0x01){ // poll PF0
        GPIO_PORTF_ICR_R = 0x01; // acknowledge flag0
        Count2++;           // event occurred
    }
}

```

*Program 2.12. Interrupt-driven edge-triggered input that counts falling edges of PF4,PF0.*

## 2.4.2. Edge-triggered Interrupts on the MSP432

Synchronizing software to hardware events requires the software to recognize when the hardware changes states from busy to done. Many times the busy to done state transition is signified by a rising (or falling) edge on a status signal in the hardware. For these situations, we connect this status signal to an input of the microcontroller, and we use edge-triggered interfacing to configure the interface to set a flag on the rising (or falling) edge of the input. Using edge-triggered interfacing allows the software to respond quickly to changes in the external world. If we are using busy-wait synchronization, the software waits for the flag. If we are using interrupt synchronization, we configure the flag to request an interrupt when set. Each of the digital I/O pins on ports P1 – P6 can be configured for edge triggering. Table 2.11 shows many of the registers available for Port 1. The differences between members of the MSP432 family include the number of ports (e.g., the MSP432P401 has ports 1 – 10), which pins can interrupt (e.g., the MSP432P401 can interrupt on ports 1 – 6) and the number of pins in each port (e.g., the MSP432P401 has pins 6 – 0 on Port 10). For more details, refer to the datasheet for your specific microcontroller.

Each of the pins on Ports 1 – 6 on the MSP432P401 can be configured as an edge-triggered input. When writing C code using these registers, include the header file for your particular microcontroller (e.g., **msp432p401r.h**). To use a pin as regular digital input or output, we clear its **SEL0** and **SEL1** bits. For regular digital input/output,



we clear **DIR** (Direction) bits to make them input, and we set **DIR** bits to make them output.

Address	7	6	5	4	3	2	1	0	Name
0x4000.4C00	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	P1IN
0x4000.4C02	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	P1OUT
0x4000.4C04	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	P1DIR
0x4000.4C06	REN	REN	REN	REN	REN	REN	REN	REN	P1REN
0x4000.4C08	DS	DS	DS	DS	DS	DS	DS	DS	P1DS
0x4000.4C0A	SEL0	SEL0	SEL0	SEL0	SEL0	SEL0	SEL0	SEL0	P1SEL0
0x4000.4C0C	SEL1	SEL1	SEL1	SEL1	SEL1	SEL1	SEL1	SEL1	P1SEL1
0x4000.4C0E				P1IV					P1IV
0x4000.4C18	IES	IES	IES	IES	IES	IES	IES	IES	P1IES
0x4000.4C1A	IE	IE	IE	IE	IE	IE	IE	IE	P1IE
0x4000.4C1C	IFG	IFG	IFG	IFG	IFG	IFG	IFG	IFG	P1IFG

**Table 2.11. MSP432 Port 1 registers. SEL0 SEL1 bits, see Table 2.3. All except PxIV are 8 bits wide.**

To configure an edge-triggered pin, we first configure the pin as a regular digital input. Most busy to done conditions are signified by edges, and therefore we trigger on edges of those signals. Next we write to the **IES** (Interrupt Edge Select) to define the active edge. We can trigger on the rising or falling edge, as listed in Table 2.12. We clear the **IE** (Interrupt Enable) bits if we are using busy-wait synchronization, and we set the **IE** bits to use interrupt synchronization. For input signals we have the option of adding either a pull-up resistor or a pull-down resistor. If we set the corresponding **REN** (Resistor Enable) bit on an input pin, we internally connect the equivalent of a 20 – 50 kΩ resistor to the pin. As previously mentioned we choose pull up by setting the corresponding bit in **P1OUT** to 1. We choose pull down by clearing the corresponding bit in **P1OUT** to 0.

The 16-bit **P1IV** (Interrupt Vector) register specifies a number of the highest priority flag that is set in the **P1IFG** register. The value is 0x00 if no flag is set. Pin 0 is the highest priority and Pin 7 is the lowest. If pin  $n$  is the highest priority flag that is set, then **P1IV** will be  $2*(n+1)$ , meaning it will be one of these values: 0x02, 0x04, 0x06, 0x08, 0x0A, 0x0C, 0x0E, or 0x10.

The hardware sets an **IFG** (Interrupt Flag) bit (called the trigger) and the software clears it (called the acknowledgement). The triggering event listed in Table 2.12 will set the corresponding **IFG** bit in the **P1IFG** register regardless of whether or not that bit is allowed to request an interrupt. In other words, clearing an **IE** bit disables the corresponding pin's interrupt, but it will still set the corresponding **IFG** bit when the interrupt would have occurred. To use interrupts, clear the **IE** bit, configure the bits in Table 2.11, and then set the **IE** bit. The software can acknowledge the event by writing zeros to the corresponding **IFG** bits in the **P1IFG** register. For example, to clear bit 2 in the **P1IFG** register, we simply execute

**P1IFG &= (~0x04);**

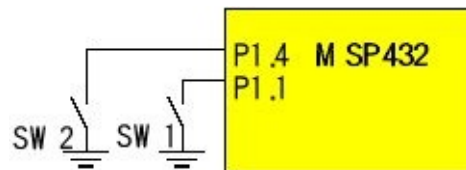
However, this mechanism has a critical section, and should not be used if there are

multiple interrupts active on a single port. The example will illustrate using **P1IV** to acknowledge.

DIR	SEL0 SEL1	IE	IES	Port mode
0	00	0	0	Input, rising edge trigger
0	00	0	1	Input, falling edge trigger
0	00	1	0	Input, rising edge trigger, interrupt
0	00	1	1	Input, falling edge trigger, interrupt

**Table 2.12. Edge-triggered modes.**

A typical application of pull-up and pull-down mode is the interface of simple switches. Using these modes eliminates the need for an external resistor when interfacing a switch. The P1.1 and P1.4 interfaces will use software-configured internal resistors. The P1.1 and P1.4 interfaces in Figure 2.7 implement negative logic switch inputs.



*Figure 2.7. Edge-triggered interfaces can generate interrupts on a switch touch. These negative logic switches require internal pull-up resistors.*

Using edge triggering to synchronize software to hardware centers around the operation of the trigger flags, **IFG**. A busy-wait interface will read the appropriate **IFG** bit over and over, until it is set. When the **IFG** bit is set, the software will clear the bit by writing a zero to it and perform the desired function. With interrupt synchronization, the initialization phase will arm the trigger flag by setting the corresponding **IE** bit. In this way, the active edge of the pin will set the **IFG** and request an interrupt. The interrupt will suspend the main program and run a special interrupt service routine (ISR). This ISR will clear the **IFG** bit and perform the desired function. At the end of the ISR it will return, causing the main program to resume. In particular, five conditions must be simultaneously true for an edge-triggered interrupt to be requested:

- The trigger flag bit is set (IFG)
- The arm bit is set (IE)
- The level of the edge-triggered interrupt must be less than **BASEPRI**
  - The edge-triggered interrupt must be enabled in the **NVIC\_ISER1**
  - Bit 0 of the special register **PRIMASK** is 0

In Volumes 1 and 2, we developed blind-cycle and busy-wait solutions. However, in this section we will redesign the systems using interrupt synchronization. Table 2.11 lists the registers for Port 1. The other ports have similar registers. However, only Ports 1 – 6 can request interrupts. We will begin with a simple example that counts the number of falling edges on Port 1 bits 1 and 4 (Program 2.13). The initialization requires many steps. We enable interrupts ( **EnableInterrupts()** ) only after all devices are initialized.

- (a) The global variables should be initialized.
- (b) The appropriate pins must be enabled as inputs.
- (c) We must specify whether to trigger on the rising or the falling edge. We will trigger on the falling of either P1.1 or P1.4. A falling edge occurs whenever we touch either SW1 or SW2.
- (d) It is good design to clear the trigger flag during initialization so that the first interrupt occurs due to the first falling edge after the initialization has been run. We do not wish to trigger on a rising edge that might have occurred during the power up phase of the system.
- (e) We arm the edge-trigger by setting the corresponding bits in the **IE** register.
- (f) We establish the priority of Port 1 by setting bits 31 – 29 in the **NVIC\_IPR8** register.
- (g) We activate Port 1 interrupts in the NVIC by setting bit 3 in the **NVIC\_ISER1** register.

The proper way to poll the interrupt is to use **P1IV**. If the software reads **P1IV** it will get the number ( $2^{*(n+1)}$ ) where  $n$  is the pin number of the lowest bit with a pending interrupt. This access will clear only flag  $n$ .

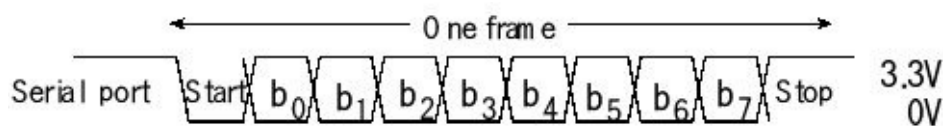
```
int32_t Count1,Count2 = 0;
void Switch_Init(void){
    Count1 = Count2 = 0;           // (a) initialize counters
    P1SEL1 &= ~0x12;              // (b) configure P1.1, P1.4 as GPIO
    P1SEL0 &= ~0x12;              //   built-in Buttons 1 and 2
    P1DIR &= ~0x12;               //   make P1.1, P1.4 in
    P1REN |= 0x12;                //   enable pull resistors
    P1OUT |= 0x12;                //   P1.1, P1.4 is pull-up
    P1IES |= 0x12;                // (c) P1.1, P1.4 is falling edge event
    P1IFG &= ~0x12;              // (d) clear flag1 and flag4
    P1IE |= 0x12;                 // (e) arm interrupt on P1.1, P1.4
    NVIC_IPR8 = (NVIC_IPR8&0x00FFFFFF)|0x40000000; // (f) priority 2
    NVIC_ISER1 = 0x00000008;     // (g) enable interrupt 35 in NVIC
}
void PORT1_IRQHandler(void){ uint8_t status;
    status = P1IV; // 4 for P1.1 and 10 for P1.4
```

```
if(status == 4){  
  Count1++;      // event occurred  
}  
if(status == 10){  
  Count2++;      // event occurred  
}  
}
```

*Program 2.13. Interrupt-driven edge-triggered input that counts falling edges of P1.4 and P1.1.*

## 2.5. UART Interface

In this section we will develop a simple device driver using the Universal Asynchronous Receiver/Transmitter (UART). This serial port allows the microcontroller to communicate with devices such as other computers, printers, input sensors, and LCDs. Serial transmission involves sending one bit a time, such that the data is spread out over time. The total number of bits transmitted per second is called the **baud rate**. The reciprocal of the baud rate is the **bit time**, which is the time to send one bit. Most microcontrollers have at least one UART. The details of the UART operation on the MSP432/TM4C can be found in Volume 2. In this book, we present general features common to all devices, and also include interrupt driven drivers. Each UART will have a baud rate control register, which we use to select the transmission rate. Each device is capable of creating its own serial clock with a transmission frequency approximately equal to the serial clock in the computer with which it is communicating. A **frame** is the smallest complete unit of serial transmission. Figure 2.8 plots the signal versus time on a serial port, showing a single frame, which includes a **start bit** (which is 0), 8 bits of data (least significant bit first), and a **stop bit** (which is 1). There is always only one start bit, but the UARTs allow us to select the 5 to 8 data bits and 1 or 2 stop bits. The UART can add even, odd, or no parity bit. However, we will employ the typical protocol of 1 start bit, 8 data bits, no parity bit, and 1 stop bit. This protocol is used for both transmitting and receiving. The information rate, or **bandwidth**, is defined as the amount of data or useful information transmitted per second. From Figure 2.8, we see that 10 bits are sent for every byte of usual data. Therefore, the bandwidth of the serial channel (in bytes/second) is the baud rate (in bits/sec) divided by 10.



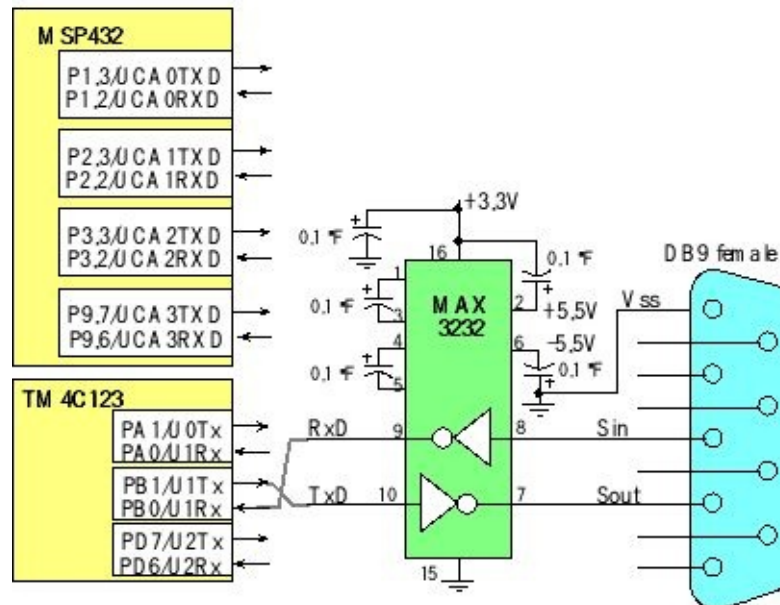
*Figure 2.8. A serial data frame with 8-bit data, 1 start bit, 1 stop bit, and no parity bit.*

**Checkpoint 2.8:** Assuming the protocol drawn in Figure 2.8 and a baud rate of 115200 bits/sec, what is the bandwidth in bytes/sec?

Table 2.13 shows the three most commonly used RS232 signals. The RS232 standard uses a DB25 connector that has 25 pins. The EIA-574 standard uses RS232 voltage levels and a DB9 connector that has only 9 pins. The most commonly used signals of the full RS232 standard are available with the EIA-574 protocols. Only **TxD**, **RxD**, and **SG** are required to implement a simple bidirectional serial channel (Figure 2.9). We define the **data terminal equipment** (DTE) as the computer or a terminal and the **data communication equipment** (DCE) as the modem or printer.

DB25 Pin	RS232 Name	DB9 Pin	EIA-574 Name	Signal	Description	True	DTE	DCE
2	BA	3	103	TxD	Transmit Data	-12V	out	in
3	BB	2	104	RxD	Receive Data	-12V	in	out
7	AB	5	102	SG	Signal Ground			

**Table 2.13.** The commonly-used signals on the RS232 and EIA-574 protocols.



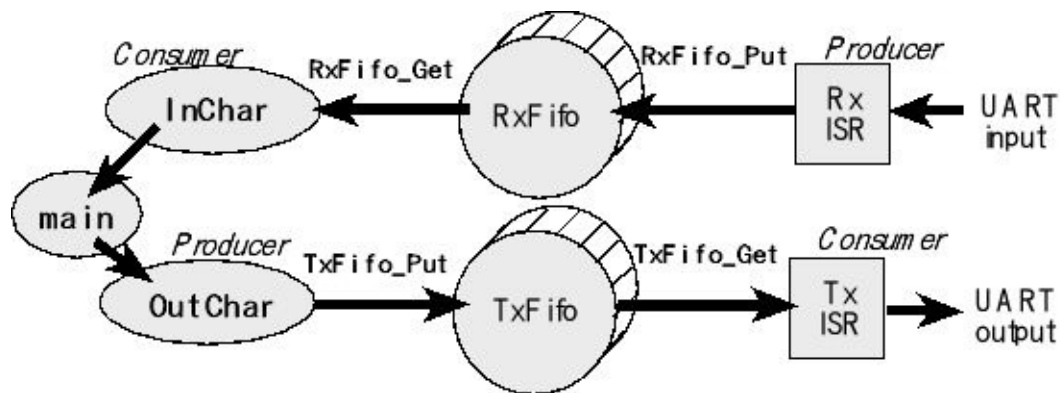
*Figure 2.9. Hardware interface implementing an asynchronous RS232 channel. The TM4C123 and TM4C1294 have eight UART ports.*

**Observation:** Most MSP432/TM4C development kits connect the UART0 channel through the USB cable, so the circuit shown in Figure 2.9 will not be needed. On the PC side of the cable, the serial channel becomes a virtual COM port.

RS232 is a non-return-to-zero (NRZ) protocol with true signified as a voltage between -5 and -15 V. False is signified by a voltage between +5 and +15 V. A MAX3232 converter chip is used to translate between the +5.5/-5.5 V RS232 levels and the 0/+3.3 V digital levels, as shown in Figure 2.9. The capacitors in this circuit are important, because they form a charge pump used to create the  $\pm 5.5$  voltages from the +3.3 V supply. The RS232 timing is generated automatically by the UART. During transmission, the Maxim chip translates a digital high on microcontroller side to -5.5V on the RS232/EIA-574 cable, and a digital low is translated to +5.5V. During receiving, the Maxim chip translates negative voltages on RS232/EIA-574 cable to a digital high on the microcontroller side, and a positive voltage is translated to a digital low. The computer is classified as DTE, so its serial output is pin 3 in the EIA-574 cable, and its serial input is pin 2 in the EIA-574 cable. When connecting a DTE to another DTE, we use a cable with pins 2 and 3 crossed. I.e., pin 2 on one DTE is connected to pin 3 on the other DTE and pin 3 on one DTE is connected to

pin 2 on the other DTE. When connecting a DTE to a DCE, then the cable passes the signals straight across. In all situations, the grounds are connected together using the SG wire in the cable. This channel is classified as **full duplex**, because transmission can occur in both directions simultaneously.

Figure 2.10 shows a data flow graph with buffered input and buffered output. First in first out (FIFO) queues are statically allocated global structures. The producer puts into the FIFO and the consumer gets from the FIFO. Because they are global variables, it means they will exist permanently and can be carefully shared by the foreground and background threads. The advantage of using a FIFO structure for a data flow problem is that we can decouple the producer and consumer threads. Without the FIFO we would have to produce one piece of data, then process it, produce another piece of data, then process it. With the FIFO, the producer thread can continue to produce data without having to wait for the consumer to finish processing the previous data. This decoupling can significantly improve system performance.



*Figure 2.10. A data flow graph showing two FIFOs that buffer data between producers and consumers.*

**Checkpoint 2.9:** What does it mean if the Rx Fifo in Figure 2.10 is empty?

**Checkpoint 2.10:** What does it mean if the Tx Fifo in Figure 2.10 is empty?

## 2.5.1. Transmitting in asynchronous mode

We will begin with transmission, because it is simple. The transmitter portion of the UART includes a data output pin, with digital logic levels as drawn in Figure 2.11. The TM4C transmitter has a 16-element FIFO and a 10-bit shift register, which cannot be directly accessed by the programmer (Figure 2.11). The MSP432 simply has the data register and shift register. The data register, FIFO, and shift register in the transmitter are separate from the data register, FIFO, and shift register associated with the receiver. To output data using the UART, the software will first check to make sure the transmit data register is not full and then write to the transmit data register (e.g., `UART0_DR_RUCA0TXBUF`). The bits are shifted out in this order: start,  $b_0$ ,  $b_1$ ,  $b_2$ ,  $b_3$ ,  $b_4$ ,  $b_5$ ,  $b_6$ ,  $b_7$ , and then stop, where  $b_0$  is the LSB and  $b_7$  is the MSB.

The transmit data register is write only, which means the software can write to it (to start a new transmission) but cannot read from it. Even though the transmit data register is at the same address as the receive data register, the transmit and receive data registers are two separate registers.

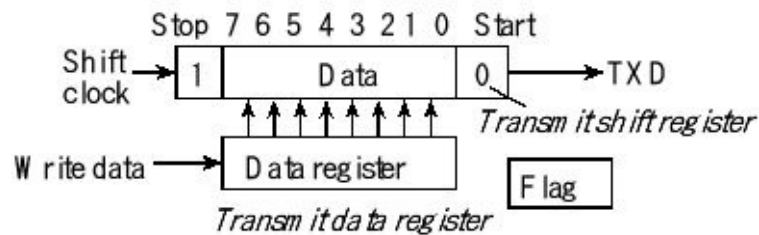


Figure 2.11. Data and shift registers implement the serial transmission.

On the TM4C, we will interrupt when the transmit FIFO is almost empty. The ISR will pass data from the software FIFO to the hardware FIFO. The use of FIFOs separates the data production (software) from the data consumption (UART hardware).

On the MSP432, we will interrupt when the transmit data register is empty. The ISR will pass one byte of data from the software FIFO to the hardware UART.

In all cases, we will disarm the UART transmitter when the software FIFO is empty, and rearm it when new data are available.

## 2.5.2. Receiving in asynchronous mode

Receiving data frames is a little trickier than transmission because we have to synchronize the receive shift register with the incoming data. The receiver portion of the UART includes an **RXD** data input pin with digital logic levels. At the input of the microcontroller, true is 3.3V and false is 0V. The TM4C microcontrollers have a 16-element FIFO to buffer the incoming frames. All microcontrollers have a 10-bit shift register and a data register. The FIFO and shift register cannot be directly accessed by the programmer (Figure 2.12). Again the receive hardware is separate from the transmitter hardware. The receive data register, **UART0\_DR\_RUCA0RXBUF**, is read only, which means write operations to this address have no effect on this register (recall write operations activate the transmitter). The receiver obviously cannot start a transmission, but it recognizes a new frame by its start bit. The bits are shifted in using the same order as the transmitter shifted them out: start, **b<sub>0</sub>**, **b<sub>1</sub>**, **b<sub>2</sub>**, **b<sub>3</sub>**, **b<sub>4</sub>**, **b<sub>5</sub>**, **b<sub>6</sub>**, **b<sub>7</sub>**, and then stop.



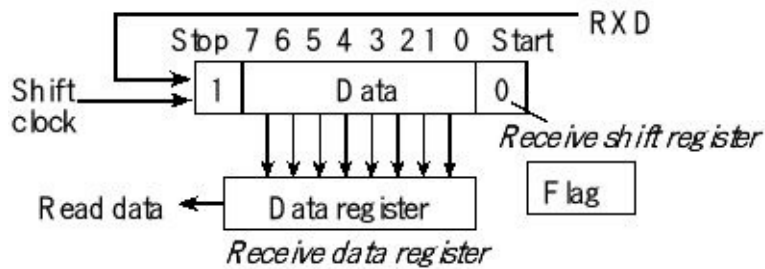


Figure 2.12. Data register shift registers implement the receive serial interface.

The receiver waits for the 1 to 0 edge signifying a start bit, then shifts in 10 bits of data one at a time from the **RXD** line. The start and stop bits are removed (checked for framing errors). The 8 bits of data are available to be read from the receive data register. On the TM4C, the FIFO implements hardware buffering so data can be safely stored if the software is performing other tasks.

We will interrupt when the receive UART has data. The ISR will pass data from the UART hardware to the software FIFO. The use of FIFOs separates the data production (UART hardware) from the data consumption (software). We will arm the UART receiver at initialization and it will remain armed throughout. If there are no incoming frames, there will be no interrupts and the software FIFO will eventually become empty. The system will remain in the idle state until new data arrives. You can find UART examples on the book web site as **UART\_XXX** and **UARTints\_XXX**.

### 2.5.3. Interrupt-driven UART on the TM4C123

The TM4C microcontrollers have one to eight UARTs. The specific port pins used to implement the UARTs vary from one chip to the next. To find which pins your microcontroller uses, you will need to consult its datasheet. Table 2.14 shows some of the registers for the UART0. If the microcontroller has multiple UARTs, the register names will replace the 0 with a 1 – 7. For the exact register addresses, you should include the appropriate header file (e.g., **tm4c1294ncpdt.h**). To activate a UART you will need to turn on the UART clock in the **SYSCTL\_RCGCUART\_R** register. You should also turn on the clock for the digital port in the **SYSCTL\_RCGCGPIO\_R** register. You need to enable the transmit and receive pins as digital signals. The alternative function for these pins must also be selected.

The OE, BE, PE, and FE are error flags associated with the receiver. You can see these flags in two places: associated with each data byte in **UART0\_DR\_R** or as a separate error register in **UART0\_RSR\_R**. The overrun error (**OE**) is set if data has been lost because the input driver latency is too long. **BE** is a break error, meaning the other device has sent a break. **PE** is a parity error (however, we will not be using parity). The framing error (**FE**) will get set if the baud rates do not match. The software can clear these four error flags by writing any value to **UART0\_RSR\_R**.

The status of the two FIFOs can be seen in the **UART0\_FR\_R** register. The **BUSY**

flag is set while the transmitter still has unsent bits. It will become zero when the transmit FIFO is empty and the last stop bit has been sent. If you implement busy-wait output by first outputting then waiting for **BUSY** to become 0, then the routine will write new data and return after that particular data has been completely transmitted.

The **UART0\_CTL\_R** control register contains the bits that turn on the UART. **TXE** is the Transmitter Enable bit, and **RXE** is the Receiver Enable bit. We set **TXE**, **RXE**, and **UARTEN** equal to 1 in order to activate the UART device. However, we should clear **UARTEN** during the initialization sequence.

	31-12	11	10	9	8	7-0				Name	
\$4000.C000		OE	BE	PE	FE	DATA				UART0_DR_R	
		31-3			3	2	1	0			
\$4000.C004					OE	BE	PE	FE		UART0_RSR_R	
	31-8	7	6	5	4	3	2-0				
\$4000.C018		TXFE	RXFF	TXFF	RXFE	BUSY				UART0_FR_R	
	31-16	15-0									
\$4000.C024		DIVINT									UART0_IBRD_R
		31-6				5-0					
\$4000.C028		DIVFRAC									UART0_FBRD_R
	31-8	7	6-5	4	3	2	1	0			
\$4000.C02C		SPS	WLEN	FEN	STP2	EPS	PEN	BRK		UART0_LCRH_R	
	31-10	9	8	7	6-3	2	1	0			
\$4000.C030		RXE	TXE	LBE		SIRLP	SIREN	UARTEN		UART0_CTL_R	
		31-6			5-3		2-0				
\$4000.C034		RXIFLSEL									UART0_IFLS_R
	31-11	10	9	8	7	6	5	4			
\$4000.C038		OEIM	BEIM	PEIM	FEIM	RTIM	TXIM	RXIM		UART0_IM_R	
\$4000.C03C		OERIS	BERIS	PERIS	FERIS	RTRIS	TXRIS	RXRIS		UART0_RIS_R	
\$4000.C040		OEMIS	BEMIS	PEMIS	FEMIS	RTMIS	TXMIS	RXMIS		UART0_MIS_R	
\$4000.C044		OEIC	BEIC	PEIC	FEIC	RTIC	TXIC	RXIC		UART0_ICR_R	

**Table 2.14. Some UART registers. Each register is 32 bits wide. Shaded bits are zero.**

The **UART0\_IBRD\_R** and **UART0\_FBRD\_R** registers specify the baud rate. The baud rate **divider** is a 22-bit binary fixed-point value with a resolution of  $2^{-6}$ . The **Baud16** clock is created from the system bus clock, with a frequency of (Bus clock

frequency)/**divider**. The baud rate is 16 times slower than **Baud16**

$$\text{Baud rate} = \text{Baud16}/16 = (\text{Bus clock frequency})/(16*\text{divider})$$

For example, if the bus clock is 8 MHz and the desired baud rate is 19200 bits/sec, then the **divider** should be  $8,000,000/16/19200$  or 26.04167. As a binary fixed-point number, this number is about 11010.000011. We can establish this baud rate by putting the 11010 into **UART0\_IBRD\_R** and the 000011 into **UART0\_FBRD\_R**. In reality, 11010.000011 is equal to  $1667/64$  or 26.046875. The baud rates in the transmitter and receiver must match within 5% for the channel to operate properly. The error for this example is 0.02%.

The three registers **UART0\_LCRH\_R**, **UART0\_IBRD\_R**, and **UART0\_FBRD\_R** form an internal 30-bit register. This internal register is only updated when a write operation to **UART0\_LCRH\_R** is performed, so any changes to the baud-rate divisor must be followed by a write to the **UART0\_LCRH\_R** register for the changes to take effect. Out of reset, both FIFOs are disabled and act as 1-byte-deep holding registers. The FIFOs are enabled by setting the **FEN** bit in **UART0\_LCRH\_R**.

To use interrupts, we will enable the FIFOs by setting the **FEN** bit in the **UART0\_LCRH\_R** register. **RXIFLSEL** specifies the receive FIFO level that causes an interrupt. **TXIFLSEL** specifies the transmit FIFO level that causes an interrupt.

**RXIFLSEL** **RX FIFO** Set **RXMIS** interrupt trigger when

0x0	≥ 1/8 full	Receive FIFO goes from 1 to 2 characters
0x1	≥ 1/4 full	Receive FIFO goes from 3 to 4 characters
0x2	≥ 1/2 full	Receive FIFO goes from 7 to 8 characters
0x3	≥ 3/4 full	Receive FIFO goes from 11 to 12 characters
0x4	≥ 7/8 full	Receive FIFO goes from 13 to 14 characters

**TXIFLSEL** **TX FIFO** Set **TXMIS** interrupt trigger when

0x0	≤ 7/8 empty	Transmit FIFO goes from 15 to 14 characters
0x1	≤ 3/4 empty	Transmit FIFO goes from 13 to 12 characters
0x2	≤ 1/2 empty	Transmit FIFO goes from 9 to 8 characters
0x3	≤ 1/4 empty	Transmit FIFO goes from 5 to 4 characters
0x4	≤ 1/8 empty	Transmit FIFO goes from 3 to 2 characters

There are seven possible interrupt trigger flags that are in the **UART0\_RIS\_R** register. The setting of the **TXRIS** and **RXRIS** flags is defined above. The **OERIS** flag is set on an overrun, new incoming frame received but the receive FIFO is full. The **BERIS** flag is set on a break error. The **PERIS** flag is set on a parity error. The **FERIS** flag is set on a framing error (stop bit is not high). The **RTRIS** is set on a receiver timeout, which is when the receiver FIFO is not empty and no incoming frames have occurred in a 32-bit time period. Each of the seven trigger flags has a corresponding arm bit in the **UART0\_IM\_R** register. A bit in the

**UART0\_MIS\_R** register set if the trigger flag is both set and armed. To acknowledge an interrupt (make the trigger flag become zero), software writes a 1 to the corresponding bit in **UART0\_IC\_R**.

The UART system has two channels, one for input and one for output, and each channel employs a separate FIFO queue. Program 2.14 shows the interrupt-driven UART device driver. During initialization, Port A pins 0 and 1 are enabled as alternate function digital signals. The two software FIFOs are initialized. The baud rate is set at 115200 bits/sec, and the hardware FIFOs are enabled. A transmit interrupt will occur as the transmit FIFO goes from 2 elements down to 1 element. Not waiting until the hardware FIFO is completely empty allows the software to refill the hardware FIFO and maintain a continuous output stream, achieving maximum bandwidth. There are two conditions that will request a receive interrupt. First, if the receive FIFO goes from 2 to 3 elements a receive interrupt will be requested. At this time there is still 13 free spaces in the receive FIFO so the latency requirement for this real-time input will be 130 bit times (about 1 ms). The other potential source of receiver interrupts is the receiver time out. This trigger will occur if the receiver becomes idle and there are data in the receiver FIFO. This trigger will allow the interface to receive input data when it comes just one or two frames at a time. In the NVIC, the priority is set at 2 and UART0 (IRQ=5) is activated. Normally, one does not enable interrupts in the individual initialization functions. Rather, interrupts should be enabled in the main program, after all initialization functions have completed.

When the main thread wishes to output it calls **UART\_OutChar**, which will put the data into the software FIFO. FIFOs will be presented in detail later in Section 4.3. Next, it copies as much data from the software FIFO into the hardware FIFO and arms the transmitter. The transmitter interrupt service will also get as much data from the software FIFO and put it into the hardware FIFO. The **copySoftwareToHardware** function has a critical section and is called by both **UART\_OutChar** and the ISR. To remove the critical section, the transmitter is temporarily disarmed in the **UART\_OutChar** function when **copySoftwareToHardware** is called. This helper function guarantees data is transmitted in the same order it was produced. When input frames are received they are placed into the receive hardware FIFO. If this FIFO goes from 2 to 3 elements, or if the receiver becomes idle with data in the FIFO, a receive interrupt occurs. The helper function **copyHardwareToSoftware** will get from the receive hardware FIFO and put into the receive software FIFO. When the main thread wished to input data it calls **UART\_InChar**. This function simply gets from the software FIFO. If the receive software FIFO is empty, it will spin.

```
#define FIFOSIZE 16 // size of the FIFOs (must be power of 2)
#define FIFOSUCCESS 1 // return value on success
#define FIFOFAIL 0 // return value on failure
AddIndexFifo(Rx, FIFOSIZE, char, FIFOSUCCESS, FIFOFAIL)
AddIndexFifo(Tx, FIFOSIZE, char, FIFOSUCCESS, FIFOFAIL)
```

```

void UART_Init(void){
    SYSCTL_RCGCUART_R |= 0x01; // activate UART0
    SYSCTL_RCGCGPIO_R |= 0x01; // activate port A
    RxFifo_Init(); TxFifo_Init(); // initialize empty FIFOs
    UART0_CTL_R &= ~UART_CTL_UARTEN; // disable UART
    UART0_IBRD_R = 3; // IBRD=int(6,000,000/(16*115,200)) = int(3.2552)
    UART0_FBRD_R = 16; // FBRD = round(0.2552 * 64) = 16
    UART0_LCRH_R = (UART_LCRH_WLEN_8|UART_LCRH_FEN); // 8-bit, FIFOs
    UART0_IFLS_R &= ~0x3F; // TX FIFO <= 1/8 full, RX FIFO >= 1/8 full
    UART0_IFLS_R += (UART_IFLS_TX1_8|UART_IFLS_RX1_8); // and RX time-out
    UART0_IM_R |= (UART_IM_RXIM|UART_IM_TXIM|UART_IM_RTIM);
    UART0_CTL_R |= 0x0301; // enable RXE TXE UARTEN
    GPIO_PORTA_AFSEL_R |= 0x03; // enable alt funct on PA1-0
    GPIO_PORTA_DEN_R |= 0x03; // enable digital I/O on PA1-0
    NVIC_PRI1_R = (NVIC_PRI1_R&0xFFFF00FF)|0x00004000; // UART0=priority 2
    NVIC_EN0_R = NVIC_EN0_INT5; // enable interrupt 5 in NVIC
    EnableInterrupts();
}
// copy from hardware RX FIFO to software RX FIFO
// stop when hardware RX FIFO is empty or software RX FIFO is full
void static copyHardwareToSoftware(void){ char letter;
    while(((UART0_FR_R&UART_FR_RXFE)==0)&&(RxFifo_Size() < (FIFOSIZE-1))){
        letter = UART0_DR_R;
        RxFifo_Put(letter);
    }
}
// copy from software TX FIFO to hardware TX FIFO
// stop when software TX FIFO is empty or hardware TX FIFO is full
void static copySoftwareToHardware(void){ char letter;
    while(((UART0_FR_R&UART_FR_TXFF) == 0) && (TxFifo_Size() > 0)){
        TxFifo_Get(&letter);
        UART0_DR_R = letter;
    }
}
// input ASCII character from UART
// spin if RxFifo is empty
char UART_InChar(void){
    char letter;
    while(RxFifo_Get(&letter) == FIFOFAIL){};
    return(letter);
}
// output ASCII character to SCI
// spin if TxFifo is full
void UART_OutChar(char data){

```

```

while(TxFifo_Put(data) == FIFOFAIL){};
UART0_IM_R &= ~UART_IM_TXIM;    // disable TX FIFO interrupt
copySoftwareToHardware();
UART0_IM_R |= UART_IM_TXIM;    // enable TX FIFO interrupt
}
// at least one of three things has happened:
// hardware TX FIFO goes from 3 to 2 or less items
// hardware RX FIFO goes from 1 to 2 or more items
// UART receiver has timed out
void UART0_Handler(void){
    if(UART0_RIS_R&UART_RIS_TXRIS){    // hardware TX FIFO <= 2 items
        UART0_ICR_R = UART_ICR_TXIC;    // acknowledge TX FIFO
        // copy from software TX FIFO to hardware TX FIFO
        copySoftwareToHardware();
        if(TxFifo_Size() == 0){        // software TX FIFO is empty
            UART0_IM_R &= ~UART_IM_TXIM;    // disable TX FIFO interrupt
        }
    }
    if(UART0_RIS_R&UART_RIS_RXRIS){    // hardware RX FIFO >= 2 items
        UART0_ICR_R = UART_ICR_RXIC;    // acknowledge RX FIFO
        // copy from hardware RX FIFO to software RX FIFO
        copyHardwareToSoftware();
    }
    if(UART0_RIS_R&UART_RIS_RTRIS){    // receiver timed out
        UART0_ICR_R = UART_ICR_RTIC;    // acknowledge receiver time out
        // copy from hardware RX FIFO to software RX FIFO
        copyHardwareToSoftware();
    }
}

```

*Program 2.14. Interrupt-driven device driver for the UART uses two FIFOs to buffer data (UARTints\_xxx).*

## 2.5.4. Interrupt-driven UART on the MSP432

Table 2.15 shows the device registers used for UART I/O. The system has two channels, one for input and one for output, and each channel employs a separate FIFO queue. Program 2.15 shows the interrupt-driven UART device driver. During initialization, Port 1 pins 2 and 3 are enabled as alternate function digital signals. The two software FIFOs are initialized. The baud rate is set at 115200 bits/sec, and the UART is enabled. A transmit interrupt will occur if the transmit data register is empty. A receive interrupt will occur if there is data in the receive data register. In the NVIC, the priority is set at 2 and the UART (eUSCI\_A, module 0, IRQ=16) is activated. Normally, one does not enable interrupts in the individual initialization

functions. Rather, interrupts should be enabled in the main program, after all initialization functions have completed.

We will employ **TXIFG** and **RXIFG** interrupt trigger flags, located in the **UCA0IFG** register. The arm bits **TXIE** and **RXIE** are located in the **UCA0IE** register. **TXIFG** is set when the **TXBUF** is empty meaning it is safe to start another output. Writing to **TXBUF** automatically clears **TXIFG**, acknowledging the transmit interrupt. **RXIFG** is set when the **RXBUF** is full meaning it is time to read the **RXBUF**. Reading **RXBUF** automatically clears **RXIFG**, acknowledging the receive interrupt. The Interrupt Enable Registers **UCAxIE** and **UCBxIE** are reset after a hardware reset or when the USCI module is in reset (bit 0 of **UCxxCTLW0** is 1).

When the main thread wishes to output it calls **UART\_OutChar**, which will put the data into the software **TxFifo**. Next, it enables the transmit interrupts. The UART ISR will copy data from the **TxFifo** to the **TXBUF**. The use of the FIFO guarantees data is transmitted in order. When the **TxFifo** becomes empty it will disarm the transmit interrupts.

	15	14	13	12	11	10	9	8	
0x40001000	PEN	PAR	MSB	7BIT	SPB	MODEx		SYNC	UCAxCTLW0
	7	6	5	4	3	2	1	0	
	SSELx		RXEIE	BRKIE	DORM	TXADDR	TXBRK	SWRST	UCAxCTLW0
	15 – 0								
0x40001006	UCBRx								UCAxBRW
	15 – 8		7 – 4			3 – 1		0	
0x40001008	BRSx			BRFx			UCOS16		UCAxMCTLW
	7	6	5	4	3	2	1	0	
0x4000100A	LISTEN	FE	OE	PE	BRK	RXERR	IDLE	BUSY	UCAxSTATW
	15 – 8		7 – 0						
0x4000100C	RXBUFx								UCAxRXBUF
	15 – 8		7 – 0						
0x4000100E	TXBUFx								UCAxTXBUF
	15 – 4				3	2	1	0	
0x4000101A	TXCPTIE				STTIE	TXIE	RXIE		UCAxIE
	15 – 4				3	2	1	0	
0x4000101C	TXCPTIFG				STTIFG	TXIFG	RXIFG		UCAxIFG

**Table 2.15. UART registers. Each register is 16 bits wide. Shaded bits are zero.**

When an input frame is received it is placed into the receive data register **RXBUF**, and a receive interrupt occurs. The ISR will read the data from **RXBUF** and put it in the software FIFO **RxFifo**. The ISR is not allowed to spin. So if **RxFifo** becomes full data are lost. When the main thread wishes to input data it calls **UART\_InChar**.

This function simply gets from the software FIFO. In contrast to the ISR, the foreground is allowed to spin. So if the main program calls **UART\_InChar** and the **RxFifo** is empty, it will spin.

```
#define FIFOSIZE 16 // size of the FIFOs (must be power of 2)
#define FIFOSUCCESS 1 // return value on success
#define FIFOFAIL 0 // return value on failure
AddIndexFifo(Rx, FIFOSIZE, char, FIFOSUCCESS, FIFOFAIL)
AddIndexFifo(Tx, FIFOSIZE, char, FIFOSUCCESS, FIFOFAIL)
void UART_Init(void){ // should be called only once
    RxFifo_Init(); // initialize FIFOs
    TxFifo_Init();
    UCA0CTLW0 = 0x0001; // hold the USCI module in reset mode
    UCA0CTLW0 = 0x00C1; // UART,SMCLK, 8bit, 1 stop,no parity, LSB first
    UCA0BRW = 26; // UCBR = baud rate = 3000000/115200 = 26.0417
    UCA0MCTLW = 0x0000; // clear first and second modulation, UCOS16=0
    P1SEL0 |= 0x0C;
    P1SEL1 &= ~0x0C; // P1.3 and P1.2 as primary module function
    NVIC_IPR4 = (NVIC_IPR4&0xFFFFF00)|0x00000040; // priority 2
    NVIC_ISER0 = 0x00010000; // enable interrupt 16 in NVIC
    UCA0CTLW0 &= ~0x0001; // enable the USCI module
    UCA0IE = 0x0001; // enable interrupts on receive full
        // disable interrupts on transmit, start, complete
} // must modify UCxxIE while USCI module not reset
// input ASCII character from UART
// spin if RxFifo is empty
char UART_InChar(void){
    char letter;
    while(RxFifo_Get(&letter) == FIFOFAIL){};
    return(letter);
}
// output ASCII character to UART
// spin if TxFifo is full
void UART_OutChar(char data){
    while(TxFifo_Put(data) == FIFOFAIL){}; // spin if full
    UCA0IE = 0x0003; // enable interrupts on transmit empty
}
// interrupt 16 occurs on either:
// UCTXIFG TX data register is empty
// UCRXIFG RX data register is full
// vector at 0x00000080 in startup_msp432.s
void EUSCIA0_IRQHandler(void){ char data;
    if(UCA0IFG&0x02){ // TX data register empty
        if(TxFifo_Get(&data) == FIFOFAIL){
```



```

UCA0IE = 0x0001;    // disable interrupts on transmit empty
}else{
UCA0TXBUF = data;    // send data, acknowledge interrupt
}
}
if(UCA0IFG&0x01){    // RX data register full
RxFifo_Put((char)UCA0RXBUF);// clears UCRXIFG
}
}

```

*Program 2.15. Interrupt-driven device driver for the UART uses two software FIFOs to buffer data (UARTint\_MSP432).*

## 2.6. Synchronous Transmission and Receiving using the SSI

SSI allows microcontrollers to communicate synchronously with peripheral devices and other microcontrollers. The SSI system can operate as a master or as a slave. The channel can have one master and one slave, or it can have one master and multiple slaves. With multiple slaves, the configuration can be a star (centralized master connected to each slave), or a ring (each node has one receiver and one transmitter, where the nodes are connected in a circle.) The master initiates all data communication. Figure 2.13 shows the I/O port locations of some of the synchronous serial ports on the Texas Instruments microcontrollers.

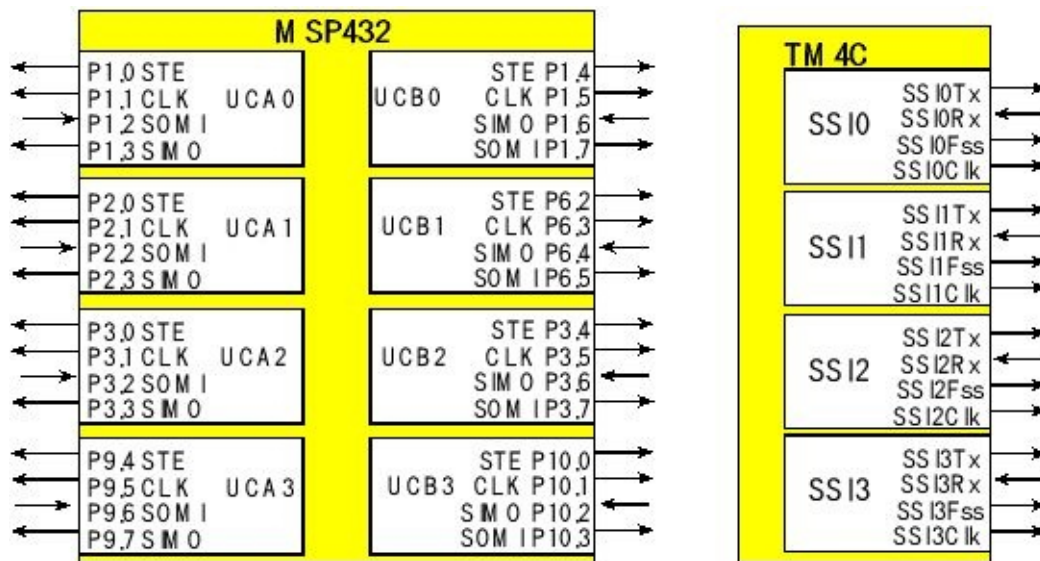


Figure 2.13. Synchronous serial port pins on four MSP432/TM4C microcontrollers.

Texas Instruments microcontrollers have 0 to 8 **Synchronous Serial Interface** or SSI modules. Another name for this protocol is **Serial Peripheral Interface** or SPI. The fundamental difference between a UART, which implements an asynchronous protocol, and a SSI, which implements a synchronous protocol, is the manner in which the clock is implemented. Two devices communicating with asynchronous serial interfaces (UART) operate at the same frequency (baud rate) but have two separate clocks. With a UART protocol, the clock signal is not included in the interface cable between devices. Two UART devices can communicate with each other as long as the two clocks have frequencies within  $\pm 5\%$  of each other. Two devices communicating with synchronous serial interfaces (SSI) operate from the same clock (synchronized). With a SSI protocol, the clock signal is included in the interface cable between devices. Typically, the master device creates the clock, and the slave device(s) uses the clock to latch the data (in or out.)

The SSI protocol includes four I/O lines. The slave select **SSI0Fss/STE** is an optional negative logic control signal from master to slave signal signifying the channel is active. The second line, **SCK/CLK**, is a 50% duty cycle clock generated by the master. The **SSI0Tx/SIMO** (master out slave in, MOSI) is a data line driven by the master and received by the slave. The **SSI0Rx/SOMI** (master in slave out, MISO) is a data line driven by the slave and received by the master. In order to work properly, the transmitting device uses one edge of the clock to change its output, and the receiving device uses the other edge to accept the data.

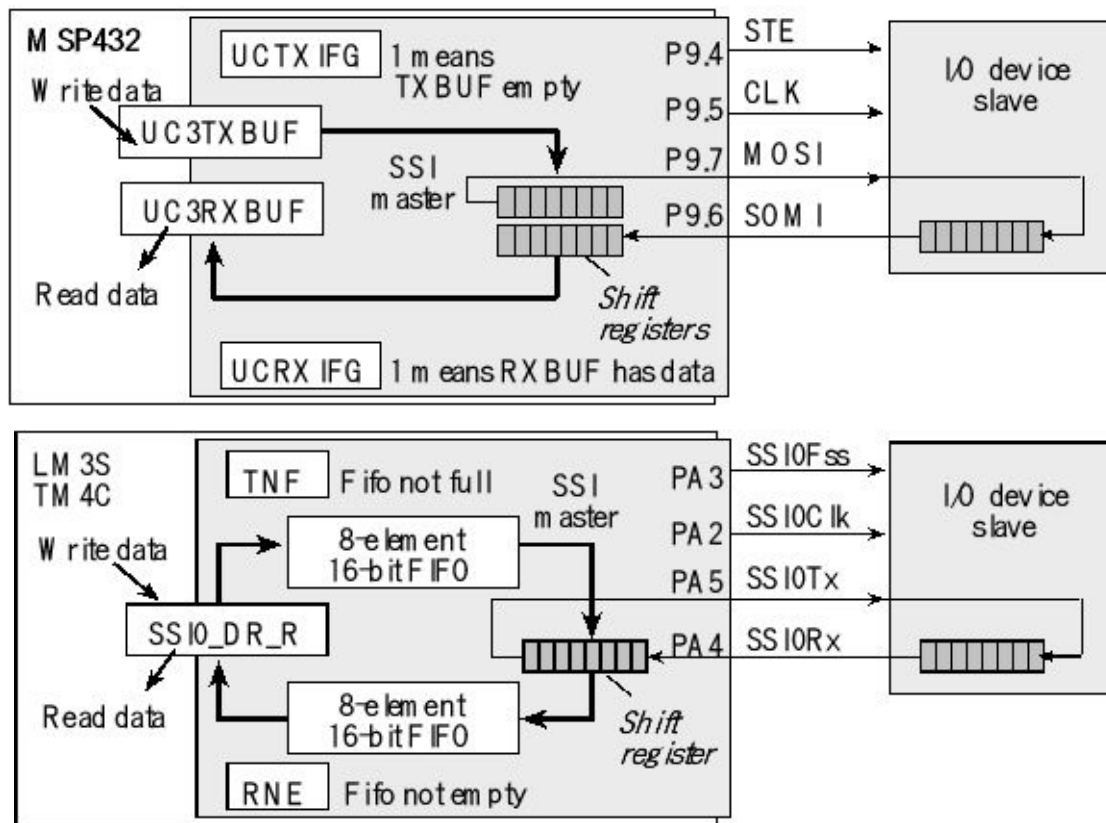


Figure 2.14. A synchronous serial interface between a microcontroller and an I/O device.

The interface is classified as synchronous because the hardware clock is shared between devices, see Figure 2.14. On the TM4C the shift register can be configured from 4 to 16 bits. On the MSP432 the shift register can be configured as 7 or 8 bits. The shift register in the master and the shift register in the slave are linked to form a distributed register. Figure 2.14 illustrates communication between master and slave. Typically, the microcontroller and the I/O device slave are so physically close we do not use interface logic. The SSI on the TM4C employs two hardware FIFOs. Both FIFOs are 8 elements deep and 4 to 16 bits wide, depending on the selected data width. When performing I/O the software puts into the transmit FIFO by writing to the **SSI0\_DR\_R/UCxTXBUF** register and gets from the receive FIFO by reading from the **SSI0\_DR\_R/UCxRXBUF** register.

When designing with SSI, you will need to consult the data sheets for your specific microcontroller. There are many SSI examples on the book web site.

---

## 2.7. Input Capture or Input Edge Time Mode

### 2.7.1. Basic principles

The Texas Instruments microcontrollers have timers that are separate and distinct from SysTick, see Figure 2.15. Input edge time mode (or input capture mode) is used to make time measurements on input signals. We can use input capture to measure the period or pulse width of digital-level signals. The input capture system can also be used to trigger interrupts on rising or falling transitions of external signals. Each timer input capture module has

- An external input pin, e.g., **CCP0/TAx.y**
- A clock, with prescale, used to measure time
- Control registers to set the mode
- Flag register that indicate status
- Arm and enable registers to implement interrupts
- A capture register, e.g., **TAR/TAxCCRy**

The various members of the MSP432/TM4C family have from zero to twenty input capture pins, and the pins are grouped into modules. Figure 2.15 shows the port pins and timer modules used for input capture on the MSP432 and TM4C123. On the TM4C, the input capture and output compare pins are labeled **TxCCPy**. On the MSP432, the input capture and output compare pins are labeled **TAx.y**. Some timer modules are not attached to any I/O pins. For example, the TM4C1294 has eight timers, but Timer 6 and Timer 7 do not have I/O pins. Timers without pins can be used to generate periodic interrupts, but not for input capture. Tables 1.4, and 1.5 describe how to attach I/O pins to the timer modules.

In this book we use the term **arm** to describe the bit that allows/denies a specific flag from requesting an interrupt. The Texas Instruments manuals refer to this bit as a **mask**. I.e., the device is armed when the mask bit is 1. Typically, there is a separate arm bit for every flag that can request an interrupt. An external input signal is connected to the input capture pin.

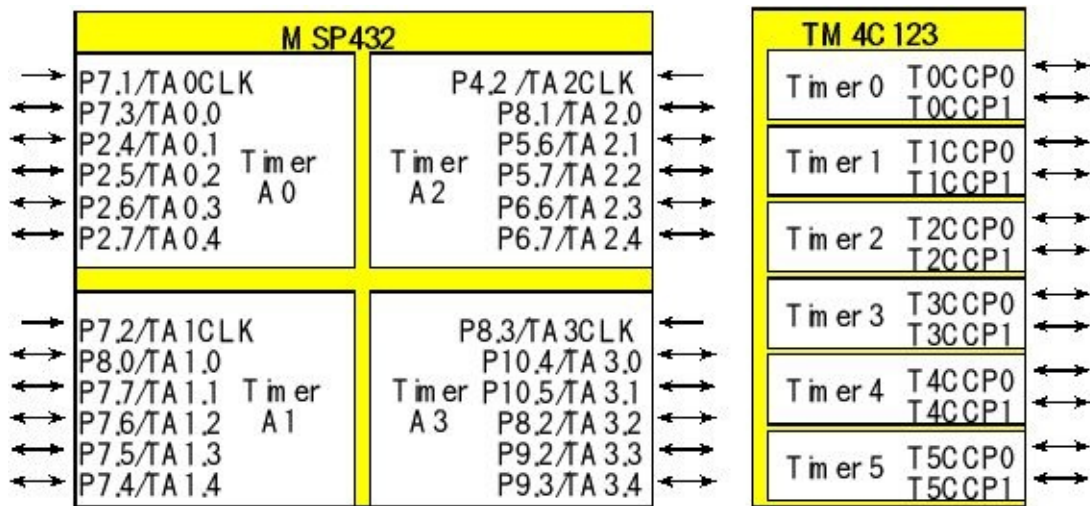


Figure 2.15. Input capture pins on the MSP432, and the TM4C123.

During initialization we specify whether the rising or falling edge of the external signal will trigger an input capture event. The timers can have 16, 24, 32, 48, or 64 bits. The  $n$ -bit counter decrements at the rate of the bus clock, when it hits 0, it automatically rolls over to all ones and continues to count down (Figure 2.16).

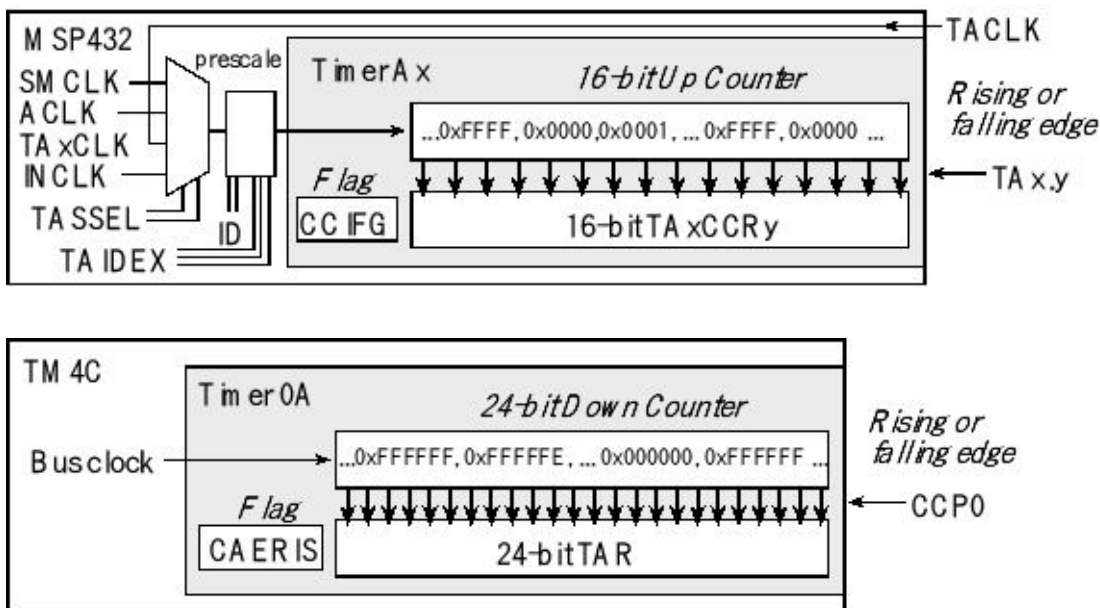


Figure 2.16. Rising or falling edge of the input causes the counter to be latched into a register, setting a flag.

Two or three actions result from an input capture event: 1) the current timer value is copied into the input capture register, 2) the input capture flag is set and 3) an interrupt is requested if armed. This means an interrupt can be requested on a capture event. When using the prescaler on the TM4C, the 16-bit counter is extended to 24 bits. The MSP432 counters are 16 bits. The input capture mechanism has many uses. Three of common applications are:

1. An ISR is executed on the active edge of the external signal
2. Perform two rising edge input captures and subtract the two

to get period

3. Perform a rising edge and then a falling edge capture and subtract the two measurements to get pulse width

## 2.7.2. Period measurement on the TM4C123

Next we will overview the specific input capture functions on the TM4C family. This section is intended to supplement rather than replace the data sheets. When designing systems with input capture, please refer to the reference manual of your specific microcontroller. Table 2.16 shows some of the registers for Timer 0. We begin initialization by enabling the clock for the timer and for the digital port we will be using. We enable the digital pin and select its alternative function. We will disable the timer during initialization by clearing the **TAEN** (or **TBEN**) bit in the **TIMER0\_CTL\_R** register. To use 16-bit mode, we set **GPTMCFG** field to 4. We clear the **TAAMS** (or **TBAMS**) bit for capture mode. We set the **TACMR** (or **TBCMR**) bit for input edge time mode. The **TAMR** (or **TBMR**) field is set to 3 for capture mode. In summary, we write a 0x0007 to the **TIMER0\_TAMR\_R** register to select input capture mode. Table 2.17 lists the edge capture modes for **TAEVENT** (or **TBEVENT**.)

When we are measuring time with prescaler, such as period measurement and pulse width measurement, we set the 24-bit reload value to 0xFFFFFFFF. In this way, the 24-bit subtraction of two capture events yields the time difference between events. In particular, we will initialize **TIMER0\_TAILR\_R** to 0xFFFF and **TIMER0\_TAPR\_R** to 0xFF. We arm the input capture by setting the **CAEIM** (or **CBEIM**) bit in the **TIMER0\_IMR\_R** register. It is good practice to clear the trigger flag in the initialization so that the first interrupt occurs due to actions occurring after the initialization, and not due to edges that might have occurred during power up. The trigger flags are in the **TIMER0\_RIS\_R** register. These flags are cleared by writing 1's into corresponding bits in the **TIMER0\_ICR\_R** register. After all configuration bits are set, the Timer can be enabled by setting the **TAEN** (or **TBEN**) bit in the **TIMER0\_CTL\_R** register. If interrupts are required, then the NVIC must be configured by setting the priority and enabling the appropriate interrupt number.

There is an 8-bit prescaler defined for each submodule A and B: **TIMER0\_TAPMR\_R** and **TIMER0\_TBPMR\_R**. The prescalers on the TM4C are used to extend the 16-bit timer to 24 bits. The **TAEVENT** bits of **TIMER0\_CTL\_R** register specify whether the rising or falling edge of **CCP0** will trigger an input capture event on Timer 0A. Two or three actions result from an input capture event: 1) the current timer value is copied into the input capture register, **TIMER0\_TAR\_R**, 2) the input capture flag (**CAERIS**) is set, and 3) an interrupt is requested if the mask bit (**CAEIM**) is 1. The **CAERIS** and **CBERIS** flag bits in the **TIMER0\_RIS\_R** register do not behave like a regular memory location.

In particular, the flag cannot be set by software. Rather, an input capture or output compare hardware event will set the flag.

	31-3					2-0			Name
\$4003.0000						<b>GPTMCFG</b>			TIMER0_CFG_R
						3	2	1-0	
\$4003.0004						<b>TAAMS</b>	<b>TACMR</b>	<b>TAMR</b>	TIMER0_TAMR_R
						3	2	1-0	
\$4003.0008						<b>TBAMS</b>	<b>TBCMR</b>	<b>TBMR</b>	TIMER0_TBMR_R
	14	13	11-10	8	6	5	3-2	0	
\$4003.000C	<b>TBPWML</b>	<b>TBOTE</b>	<b>TBEVENT</b>	<b>TBEN</b>	<b>TAPWML</b>	<b>TAOTE</b>	<b>TAEVENT</b>	<b>TAEN</b>	TIMER0_CTL_R
	31-11	10	9	8	7-4	2	1	0	
\$4003.0018		<b>CBEIM</b>	<b>CBMIM</b>	<b>TBTOIM</b>		<b>CAEIM</b>	<b>CAMIM</b>	<b>TATOIM</b>	TIMER0_IMR_R
	31-11	10	9	8	7-4	2	1	0	
\$4003.001C		<b>CBERIS</b>	<b>CBMRIS</b>	<b>TBTORIS</b>		<b>CAERIS</b>	<b>CAMRIS</b>	<b>TATORIS</b>	TIMER0_RIS_R
	31-11	10	9	8	7-4	2	1	0	
\$4003.0020		<b>CBEMIS</b>	<b>CBMMIS</b>	<b>TBTOMIS</b>		<b>CAEMIS</b>	<b>CAMMIS</b>	<b>TATOMIS</b>	TIMER0_MIS_R
	31-11	10	9	8	7-4	2	1	0	
\$4003.0020		<b>CBECINT</b>	<b>CBMCINT</b>	<b>TBTCINT</b>		<b>CAECINT</b>	<b>CAMCINT</b>	<b>TATOCINT</b>	TIMER0_ICR_R
	31-16					15-0			
\$4003.0028	<b>TAILRH</b>					<b>TAILRL</b>			TIMER0_TAILR_R
	31-16					15-0			
\$4003.002C						<b>TBILRL</b>			TIMER0_TBILR_R
	31-16					15-0			
\$4003.0030	<b>TAMRH</b>					<b>TAMRL</b>			_TAMATCHR_R
	31-16					15-0			
\$4003.0034						<b>TBMRL</b>			_TBMATCHR_R
	31-8					7-0			
\$4003.0038						<b>TAPSR</b>			TIMER0_TAPR_R
	31-8					7-0			
\$4003.003C						<b>TBPSR</b>			TIMER0_TBPR_R
	31-8					7-0			
\$4003.0040						<b>TAPSMR</b>			TIMER0_TAPMR_F
	31-8					7-0			
\$4003.0044						<b>TBPSMR</b>			TIMER0_TBPMR_F

	31-16	15-0	
\$4003.0048	TARH	<b>TARL</b>	TIMER0_TAR_R
	31-16	15-0	
\$4003.004C		TBRL	TIMER0_TBR_R

**Table 2.16. Timer0 registers. Each register is 32 bits wide. Shaded bits are zero. The bits shown in bold will be used in this section. Timers 1, 2, ... have the same formats.**

The other peculiar behavior of the flag is that the software must write a one to the **TIMER0\_ICR\_R** register in order to clear the flag. If the software writes a zero to the **TIMER0\_ICR\_R** register, no change will occur. From Table 2.16, we see the **CAERIS** trigger flag is in bit 2 of the **TIMER0\_RIS\_R** register. The proper way to clear this trigger flag is

**TIMER0\_ICR\_R = 0x0004;**

Writes the **TIMER0\_RIS\_R** register have no effect. No effect occurs in the bits to which we write a zero in the **TIMER0\_ICR\_R** register.

<b>TAEVENT</b>	Active edge
00	Capture on rising
01	Capture on falling
10	Reserved
11	Capture on both rising and falling

**Table 2.17. Two control bits define the active edge used for input capture (TBEVENT is the same).**

Before one implements a system that measures period, it is appropriate to consider the issues of resolution, precision and range. The **resolution** of a period measurement is defined as the smallest change in period that can reliably be detected. In the following example, the TM4C123 bus clock is 80 MHz. This means, if the period increases by 12.5 ns, then there will be one more Timer clock between the first rising edge and the second rising edge. In this situation, the 24-bit subtraction will increase by 1, therefore the period measurement resolution is 12.5 ns. The resolution is the smallest measurable change. Resolution defines the units of the measurement. In this first example, if the calculation of **Period** results in 1000, then it represents a period of 1000•12.5ns or 12.5µs. The **precision** of the period measurement is defined as the number of separate and distinguishable measurements. If the 24-bit counter is used, there are about 16 million different periods that can be measured. We can specify the precision in alternatives, e.g., 2<sup>24</sup>, or in bits, e.g., 24 bits. The last issue to consider is the **range** of the period measurement, which is defined as the minimum and maximum values that can reliably be measured. We are concerned what happens if the period is too small or too large. A good measurement system should be able to detect overflows and underflows. In addition, we would not like the system to crash, or



hang-up if the input period is out of range. Similarly, it is desirable if the system can detect when there is no period. For edge detection, the input must be high for at least two system clock periods and low for at least two system clock periods.

In this example, the digital input signal is connected to an input capture pin. If the motor shaft rotates once there will be  $N$  rising edges on the pin. Each rising edge will cause an input capture interrupt (Figure 2.17).

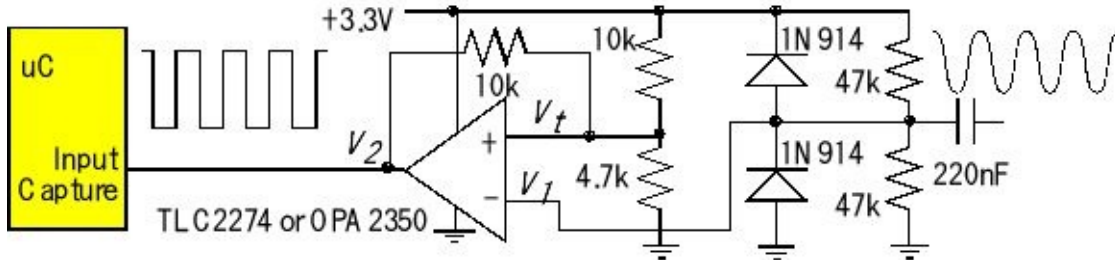


Figure 2.17. To measure period we connect the external signal an input capture.

The period is calculated as the difference in **TIMER0\_TAR\_R** latch values from one rising edge to the other. If  $N=100$ , and the motor is spinning at 300 RPM, then the period will be  $[(60000\text{ms}/\text{min})/(300\text{RPM})/100\text{edges}/\text{rotation}]$ , which will be 2.00 ms/edge, as shown in Figure 2.18.

For example, if the period is 2000  $\mu\text{s}$ , the Timer0A interrupts will be requested every 160,000 cycles, and the 24-bit difference between **TIMER0\_TAR\_R** latch values will be 160,000. This subtraction remains valid even if the timer reaches zero and wraps around in between Timer0A interrupts. On the other hand, this method will not operate properly if the period is larger than  $2^{24}$  cycles, or about 209 ms.

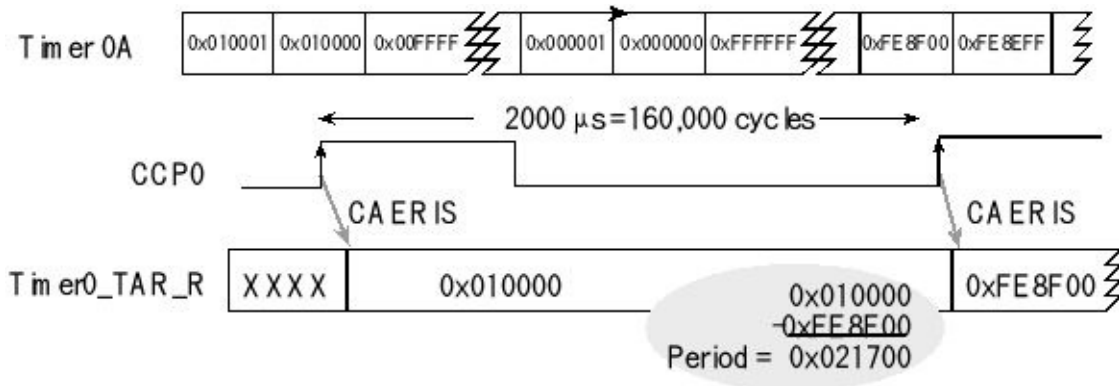


Figure 2.18. Timing example showing counter rollover during 24-bit period measurement.

The resolution is 12.5 ns because the period must increase by at least this amount before the difference between Timer0A measurements will reliably change. Even though a 24-bit counter is used, the precision is a little less than 24 bits, because the shortest period that can be handled with this interrupt-driven approach is about 1  $\mu\text{s}$ . It takes about 1  $\mu\text{s}$  to complete the context switch, execute the ISR software, and return from interrupt. This factor is determined by experimental measurement. In

other words, as the period approaches 1  $\mu$ s, a higher and higher percentage of the computer execution is utilized just in the handler itself. For example, if you wanted to limit execution time in this ISR to 5%, then the shorted period you could measure would be 20  $\mu$ s.

Because the input capture interrupt has a separate vector the software does not poll. An interrupt is requested on each rising edge of the input signal. In this situation we count all the cycles required to process the interrupt. The period measurement system written for the TM4C123 is presented in Program 2.16. The 24-bit subtraction is produced by ANDing the difference with 0x0FFFFFFF, calculating the number of bus clocks between rising edges. The first period measurement will be incorrect and should be neglected.

```

uint32_t Period;           // 24-bit, 12.5 ns units
uint32_t static First;    // Timer0A first edge, 12.5 ns units
int32_t Done;            // mailbox status set each rising
void PeriodMeasure_Init(void){
    SYSCTL_RCGCTIMER_R |= 0x01;    // activate timer0
    SYSCTL_RCGCGPIO_R |= 0x02;    // activate port B
    First = 0;                // first will be wrong
    Done = 0;                // set on subsequent
    GPIO_PORTB_DIR_R &= ~0x40;    // make PB6 input
    GPIO_PORTB_AFSEL_R |= 0x40;  // enable alt funct on PB6
    GPIO_PORTB_DEN_R |= 0x40;    // configure PB6 as T0CCP0
    GPIO_PORTB_PCTL_R = (GPIO_PORTB_PCTL_R&0xF0FFFFFF)+0x07000000;
    TIMER0_CTL_R &= ~0x00000001;  // disable timer0A during setup
    TIMER0_CFG_R = 0x00000004;    // configure for 16-bit capture mode
    TIMER0_TAMR_R = 0x00000007;  // configure for rising edge event
    TIMER0_CTL_R &= ~0x0000000C;  // rising edge
    TIMER0_TAILR_R = 0x0000FFFF;  // start value
    TIMER0_TAPR_R = 0xFF;        // activate prescale, creating 24-bit
    TIMER0_IMR_R |= 0x00000004;  // enable capture match interrupt
    TIMER0_ICR_R = 0x00000004;   // clear timer0A capture match flag
    TIMER0_CTL_R |= 0x00000001;  // timer0A 24-b, +edge, interrupts
    NVIC_PRI4_R = (NVIC_PRI4_R&0x00FFFFFF)|0x40000000; //Timer0A=priority 2
    NVIC_EN0_R = 1<<19;        // enable interrupt 19 in NVIC
    EnableInterrupts();
}
void Timer0A_Handler(void){
    TIMER0_ICR_R = 0x00000004;  // acknowledge timer0A capture
    Period = (First - TIMER0_TAR_R)&0x00FFFFFF; // 12.5ns resolution
    First = TIMER0_TAR_R;       // setup for next
    Done = 1;                    // set semaphore
}

```

## 2.7.3. Period measurement on the MSP432

Next we will overview the specific input capture functions on the MSP432 family. This section is intended to supplement rather than replace the data sheets. When designing systems with input capture, please refer to the reference manual of your specific microcontroller. Table 2.18 shows the registers for Timer A0. Similar registers are available for the A1, A2, and A3 timers. The first decision is to select a clock using the **TASSEL** bits. When measuring frequency or counting events we can connect an input signal to **TAxCLK** and use this input to count the counter. We will use **ACLK** when measuring times on the order of seconds or minutes. On the MSP432, the **ACLK** can be 10 kHz, 32.768 kHz, or 100 kHz. We will use the high speed **SMCLK** for most examples in this book because it provides the best time resolution. The **INCLK** is an internal signal that could be selected. One example of **INCLK** is the analog comparator, where a clock edge is generated when an analog input crosses a predefined threshold. Table 2.19 shows how to select the timer clock, which affects measurement resolution.

The second decision is to specify the prescaler. The first prescale is **ID**, see Table 2.20. The second prescale is **TAIDEX+1**. When measuring time events like period and pulse width, the resolution of the measurement is the period of the selected clock, *T*, multiplied by the prescale.

$$\text{Resolution} = T * 2^{\text{ID}} * (\text{TAIDEX}+1)$$

	15-10		9-8		7-6		5-4		3		2		1		0		Name
\$4000.0000			<b>TASSEL</b>		<b>ID</b>		<b>MC</b>				<b>TACLRL</b>		TAIE		TAIFG		TA0CTL
	15-14	13-12	11	10	9	8	7-5	4	3	2	1	0					
\$4000.0002	<b>CM</b>	<b>CCIS</b>	<b>SCS</b>	SCCI		<b>CAP</b>	OUTMOD	<b>CCIE</b>	CCI	OUT	COV	<b>CCIFG</b>					TA0CTL0
\$4000.0004	CM	CCIS	SCS	SCCI		CAP	OUTMOD	CCIE	CCI	OUT	COV	CCIFG					TA0CTL1
\$4000.0006	CM	CCIS	SCS	SCCI		CAP	OUTMOD	CCIE	CCI	OUT	COV	CCIFG					TA0CTL2
\$4000.0008	CM	CCIS	SCS	SCCI		CAP	OUTMOD	CCIE	CCI	OUT	COV	CCIFG					TA0CTL3
\$4000.000A	CM	CCIS	SCS	SCCI		CAP	OUTMOD	CCIE	CCI	OUT	COV	CCIFG					TA0CTL4
\$4000.000C	CM	CCIS	SCS	SCCI		CAP	OUTMOD	CCIE	CCI	OUT	COV	CCIFG					TA0CTL5
\$4000.000E	CM	CCIS	SCS	SCCI		CAP	OUTMOD	CCIE	CCI	OUT	COV	CCIFG					TA0CTL6
	15-0																
\$4000.0010	<b>16-bit counter</b>																TA0R
\$4000.0012	<b>16-bit Capture/Compare 0 Register</b>																TA0CCR0
\$4000.0014	16-bit Capture/Compare 1 Register																TA0CCR1
\$4000.0016	16-bit Capture/Compare 2 Register																TA0CCR2
\$4000.0018	16-bit Capture/Compare 3 Register																TA0CCR3
\$4000.001A	16-bit Capture/Compare 4 Register																TA0CCR4
\$4000.001C	16-bit Capture/Compare 5 Register																TA0CCR5

\$4000.001E	16-bit Capture/Compare 6 Register		TA0CCR6
	15-3	2-0	
\$4000.0020		<b>TAIDEX</b>	TA0EX0
	15-0		
\$4000.002E	TAIV		TA0IV

**Table 2.18. Timer A0 registers. Each register is 16 bits wide. Shaded bits are reserved. The bits shown in bold will be used in this section. Timers 1, 2, and 3 have the same formats.**

TASSEL	Selected Clock
00	TAxCLK
01	ACLK
10	SMCLK
11	INCLK

**Table 2.19. Two TASSEL bits specify the clock used to count the counter.**

ID	Prescale
00	/1
01	/2
10	/4
11	/8

**Table 2.20. Two ID bits specify the first prescaler which can be used to slow down the clock.**

The largest elapsed time we can measure will be the resolution times 65536 (size of the counter). For example, using **ACLK** counting at 10 kHz with a /64 prescale, the resolution will be 6.4 ms, the 16-bit counter will roll over after 7 minutes.

The **MC** bits specify the clock mode, as shown in Table 2.21. We will use “up mode” to create periodic interrupts. We will use “continuous mode” when measuring period or pulse width. In this mode the counter keeps track of time and the input edge on **TAx.y** latches the current time into the **TAxCCRy** register. We will use “up/down mode” to create PWM outputs.

MC	Mode control
00	Stop
01	Up mode: Timer counts up to <b>TAxCCR0</b>
10	Continuous mode: Timer counts up to 0xFFFF
11	Up/down mode: Timer counts up to <b>TAxCCR0</b> then down to 0x0000

**Table 2.21. Two ID bits specify the first prescaler which can be used to slow down the**

clock.

Writing a 1 to the **TACLR** bit will reset the timer and automatically clear the **TACLR** bit. The **TAIFG** flag bit is set when the timer rolls over. Its associated arm bit is **TAIE**. To clear this interrupt trigger, the software writes a 0 to **TAIFG**.

As mentioned earlier for each timer there are seven associated submodules. Five of the submodules have a pin that could be used as an input to measure time events or as an output to generate waveforms. Table 2.22 lists the three choices for selecting the edge that will cause an input capture event. A capture event copies the **TAxR** counter into **TAxCCRy** register and sets the **CCIFG** flag. If armed (**CCIE**) this flag will interrupt. To acknowledge the interrupt, the software writes a zero into the flag. These are the steps to configure an input capture:

- 1) Connect the input signal to one of the **TAx.y** timer pins
- 2) Specify the timer function in its **PxSEL1** and **PxSEL0** register
- 3) Specify it as an input by clearing the direction bit in **PxDIR**
- 4) Halt the timer during initialization (**MC=00**)
- 5) Select the clock source and prescaler
- 6) Specify the rising, falling or both edges in the **CM** bits (Table 2.22)
  - Set **CCIS** to 00 to select the input pin
  - Set **SCS** to 1 to synchronize input pin to the clock (prevents glitches)
  - Set **CAP** to 1 for capture mode
  - Set **CCIE** to arm the **CCIFG** capture flag
- 7) Set the interrupt priority in the NVIC
- 8) Arm the interrupt in the NVIC
- 9) Reset and start the timer, placing it in continuous mode

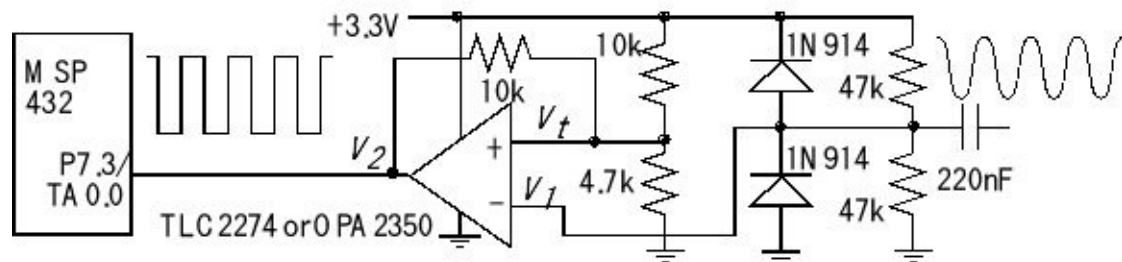
<b>CM</b>	Capture mode
00	No capture
01	Capture on rising edge
10	Capture on falling edge
11	Capture on both rising and falling edges

**Table 2.22. Two CM bits specify which edge on the TAx.y input causes the input capture.**

The basic idea of period measurement is to generate two input captures on the same edge (both rise or both fall), record the times of each edge, and calculate period as the difference between those two times. Before one implements a system that measures period, it is appropriate to consider the issues of resolution, precision and range. The **resolution** of a period measurement is defined as the smallest change in period that can reliably be detected. In Example 6.2, the **SMCLK** clock is 12 MHz.

This means, if the period increases by 83.3 ns, then there will be one more Timer clock between the first rising edge and the second rising edge. In this situation, the 16-bit subtraction will increase by 1, therefore the period measurement resolution is 83.3 ns. The resolution is the smallest measurable change. Resolution defines the units of the measurement. In this first example, if the calculation of **Period** results in 1000, then it represents a period of  $1000 \cdot 83.3\text{ns}$  or  $83.3\mu\text{s}$ . The **precision** of the period measurement is defined as the number of separate and distinguishable measurements. If the 16-bit counter is used, there are about 65,536 different periods that can be measured. We can specify the precision in alternatives, e.g.,  $2^{16}$ , or in bits, e.g., 16 bits. The last issue to consider is the **range** of the period measurement, which is defined as the minimum and maximum values that can reliably be measured. We are concerned what happens if the period is too small or too large. A good measurement system should be able to detect overflows and underflows. In addition, we would not like the system to crash, or hang-up if the input period is out of range. Similarly, it is desirable if the system can detect when there is no period. For edge detection, the input must be high for at least two system clock periods and low for at least two timer clock periods.

In this example, the digital input signal is connected to an input capture pin, P7.3/TA0.0. The diodes, 47k, and 220nF create a 0 to 3.3V signal on  $V_1$ . The 10k-4.7k create a reference voltage  $V_t$ , and the 10k positive feedback resistor removes glitches.  $V_2$  is a squarewave at the same frequency as the input. Let  $N$  be the number of rising edges as the shaft rotates once. We will set the timer period to  $5.33\mu\text{s}$ . Each rising edge will cause Timer A0 to generate an input capture interrupt (Figure 2.19).



*Figure 2.19. To measure period, we connect the external signal an input capture, P7.3 on the MSP432.*

The period is calculated as the difference in **TA0CCR0** latch values from one rising edge to the other. If  $N=100$ , and the motor is spinning at 300 RPM, then the period will be  $[(60000\text{ms}/\text{min}) / (300\text{RPM}) / 100\text{edges}/\text{rotation}]$ , which will be 2.00 ms/edge, see Figure 2.20.

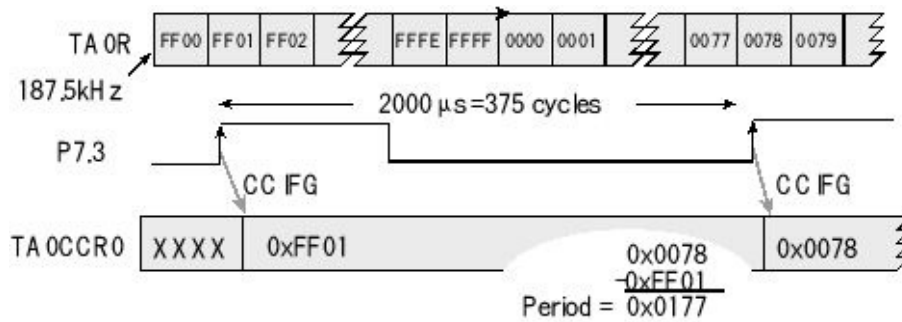


Figure 2.20. Timing example showing counter rollover during 16-bit period measurement.

For example, if the period is 2000 μs, the capture interrupts will be requested every 2 ms, which will be every  $2000/5.333 = 375$  timer clocks. The 16-bit difference between TA0CCR0 latch values will be 375. This subtraction remains valid even if the timer reaches 0xFFFF and wraps around in between interrupts. On the other hand, this method will not operate properly if the period is larger than  $2^{16}$  timer clock periods, or about 349 ms.

The resolution is 5.33μs because the period must increase by at least this amount before the difference between Timer A0 measurements will reliably change. Even though a 16-bit counter is used, the precision is a little less than 16 bits, because the shortest period that can be handled with this interrupt-driven approach is about 10 μs. It takes on the order of 10 μs to complete the context switch, execute the ISR software, and return from interrupt. This factor is determined by experimental measurement. In other words, as the period approaches 10 μs, a higher and higher percentage of the computer execution is utilized just in the handler itself.

Because the TA0.0 input capture interrupt has a separate vector the software does not poll. An interrupt is requested on each rising edge of the input signal. In this situation we count all the cycles required to process the interrupt. The period measurement system written for the MSP432 is presented in Program 2.17. The 16-bit subtraction is produced by defining the variables as 16-bit unsigned integers. The first period measurement will be incorrect and should be neglected.

```

uint16_t Period;           // 16-bit, 5.33us units
uint16_t static First;    // Timer A0 first edge, 5.33us units
int32_t Done;             // mailbox status set each rising
void PeriodMeasure_Init(void){
    Clock_Init48MHz(); // 48 MHz bus clock; 12 MHz SMCLK
    P7SEL0 |= 0x08; // 2) configure P7.3 as TA0CCP0
    P7SEL1 &= ~0x08;
    P7DIR &= ~0x08; // 3) make P7.3 in
    TA0CTL &= ~0x0030; // 4) halt Timer A0
    TA0CTL = 0x02C0; // 5) SMCLK, divide by 8
    TA0EX0 |= 0x0007; // clock divide by 8, 12MHz/64 = 187.5kHz
    TA0CCTL0 = 0x4910; // 6) rising, capture, sync, arm

```

```

NVIC_IPR2 = (NVIC_IPR2&0xFFFFF00)|0x00000040; // 7) priority 2
NVIC_ISER0 = 0x00000100; // 8) enable interrupt 8 in NVIC
TA0CTL |= 0x0024; // 9) reset and start in continuous mode
EnableInterrupts();
}
void TA0_0_IRQHandler(void){
    TA0CCTL0 &= ~0x0001; // acknowledge TA0.0 capture
    Period = TA0CCR0 - First; // 5.33us resolution
    First = TA0CCR0; // setup for next
    Done = 1; // set semaphore
}

```

*Program 2.17. 16-bit period measurement (PeriodMeasure\_MSP432).*

## 2.7.4. Pulse width measurement

The basic idea of pulse width measurement is to cause an input capture event on both the rising and falling edges of an input signal. Each edge captures a timer value. The difference between these two captured times will be the pulse width. Just like period measurement, the resolution is determined by the rate at which the timer is decremented. The maximum pulse width is  $2^{24}$  times the resolution, and is limited by the 24-bit timer.

The difficulty with pulse width measurement using one timer is the need to switch from rising to falling edge during each measurement. However, to handle shorter pulses we will need to use two input capture pins. One pin measures the time of the rise and the other pin measures the time of the fall. In order for input capture to operate, the input must be high for at least two bus clocks and low for at least two bus clocks. Otherwise the minimum pulse width does not depend on software execution time or interrupt latency. However, the minimum period will depend on software speed.

## 2.7.5. Ultrasonic distance measurement

One method to measure the distance between two objects is to transmit an ultrasonic wave from one object at the other and listen for the reflection (Figure 2.21). The instrument must be able to generate the sound pulse, hear the echo and measure the time,  $t_{in}$ , between pulse and echo. If the speed of sound,  $c$ , is known, then the distance,  $d$ , can be calculated. Our microcontrollers also have mechanisms to measure the pulse width  $t_{in}$ .

$$d = c t_{in} / 2$$



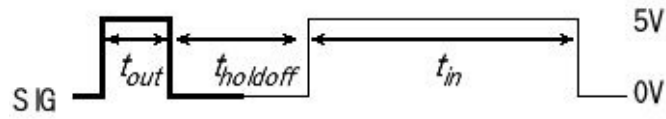
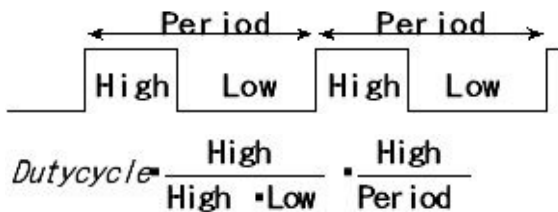


Figure 2.21. An ultrasonic pulse-echo transducer measures the distance to an object, Ping))).

## 2.8. Pulse Width Modulation

Generating output waves is an essential task for real-time systems, so the microcontrollers have multiple methods to create output waves. **Pulse width modulation** (PWM) is an effective and thus popular mechanism for the embedded microcontrollers to control external devices. Typically, the period of a PWM output is fixed, and the duty cycle is varied. The output is one for **High** cycles and then zero for **Low** cycles. To make the period constant we will configure it so **High+Low** is a constant.



### 2.8.1. Pulse width modulation on the TM4C123

PWM outputs are so important, the TM4C has a dedicated PWM modules. The number of PWMs and associated pins vary from one microcontroller to the next, see Figure 2.22.

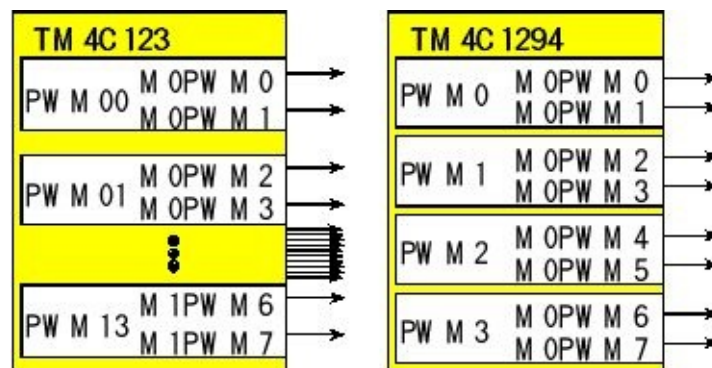


Figure 2.22. PWM pins. The TM4C123 has two PWM modules, each with four PWM generator blocks and a control block (sixteen total outputs), and the TM4C1294 has one PWM module, with four PWM generator blocks and a control block (eight total outputs).

The PWM0 block produces the PWM0 and PWM1 outputs, the PWM1 block produces the PWM2 and PWM3 outputs, and the PWM2 block produces the PWM4 and PWM5 outputs. The design of a PWM system considers three factors. The first factor is period of the PWM output. Most applications choose a period, initialize the waveform at that period, and adjust the duty cycle dynamically. The second factor is

precision, which is the total number of duty cycles that can be created. A 16-bit channel can potentially create up to 65536 different duty cycles. However, since the duty cycle register must be less than or equal to the period register, the precision of the system is determined by the value written to the period register. The last consideration is the number of channels. Different members of the TM4C family have from zero to sixteen PWM outputs (refer to the data sheet for your specific microcontroller.)

Program 2.18 shows the initialization on a TM4C123 for generating a PWM on the PB6/PWM0A pin. 1) First, we activate the clock for the PWM module. 2) Second, we activate the output pin as a digital alternate function. 3) Next, we select the clock to be used for the PWM in RCC register. If we do not use the PWM divider, then it is clocked from the bus clock. With the divider we can choose /2, /4, /8, /16, /32, or /64. If the TM4C123 is running at 50 MHz, this program specifies the PWM clock to be 25 MHz. 4) We set the PWM to countdown mode. We specify in the **PWM\_0\_GENA\_R** register that the comparator action is to set to one, and the load action is set to zero. 5) We specify the period in the **PWM\_0\_LOAD\_R** register. 6) We specify the duty cycle in the **PWM\_0\_CMPA\_R** register. 7) Lastly, we start and enable the PWM.

We call **PWM0A\_Init** once to turn it on, and then call **PWM0A\_Duty** to adjust the duty cycle. Assume the bus clock is 50 MHz, we call **PWM0A\_Init(25000,12500)**; to create a 1 ms period 50 % duty cycle output on PWM0A (PB6).

```
// period is 16-bit number of PWM clock cycles in one period (3<=period)
// duty is number of PWM clock cycles output is high (2<=duty<=period-1)
// PWM clock rate = processor clock rate/SYSCTL_RCC_PWMDIV
//          = BusClock/2
void PWM0A_Init(uint16_t period, uint16_t duty){
    SYSCTL_RCGCPWM_R |= 0x00000001; // 1) activate clock for PWM0
        // allow time to finish activating
    while((SYSCTL_PRPWM_R&0x00000001)==0){};
    SYSCTL_RCGCGPIO_R |= 0x00000002; // activate clock for Port B
        // allow time to finish activating
    while((SYSCTL_PRGPIO_R&0x00000002)==0){};
    GPIO_PORTB_AFSEL_R |= 0x40; // 2) enable alt funct on PB6
    GPIO_PORTB_ODR_R &= ~0x40; // disable open drain on PB6
    GPIO_PORTB_DEN_R |= 0x40; // enable digital I/O on PB6
    GPIO_PORTB_AMSEL_R &= ~0x40; // disable analog function on PB6
        // configure PB6 as PWM
    GPIO_PORTB_PCTL_R = (GPIO_PORTB_PCTL_R&0xF0FFFFFF)+0x04000000;
    SYSCTL_RCC_R = 0x00100000 | // 3) use PWM divider
        ((SYSCTL_RCC_R & (~0x000E0000)) + // clear PWM divider field
        0x00000000); // configure for /2 divider
    PWM0_0_CTL_R = 0; // 4) re-loading down-counting mode
        // PB6 goes low on LOAD
}
```

```

PWM0_0_GENA_R = 0x000000C8; // PB6 goes high on CMPA down
PWM0_0_LOAD_R = period - 1; // 5) cycles needed to count down to 0
PWM0_0_CMPA_R = duty - 1; // 6) count value when output rises
PWM0_0_CTL_R |= 0x00000001; // 7) start PWM0 Generator 0
PWM0_ENABLE_R |= 0x00000001; // enable PWM0 Generator 0
}
// change duty cycle
// duty is number of PWM clock cycles output is high (2<=duty<=period-1)
void PWM0A_Duty(uint16_t duty){
    PWM0_0_CMPA_R = duty - 1; // 6) count value when output rises
}

```

*Program 2.18. Implementation of a 16-bit PWM output (PWM\_XXX).*

## 2.8.2. Pulse width modulation on the MSP432

On the MSP432 each Timer A module can create one to four PWM outputs by using submodule 0 to define the period and using one to four of the other submodules to create the output and set the duty cycle. In this example Timer A0 is set to up/down mode. PWM outputs can also be created with up mode, but in this section we will describe up/down mode.

In this example, we will set **TA0CCR0** to 10, and **TA0CCR1** to 7 creating a 70% duty cycle PWM output on **P2.4/TA0.1**. In up/down mode, the **TA0R** timer will count 0, 1, 2, ... 9, 10, 9, ..., 2, 1, 0, 1, 2, ... over and over. We will use toggle/reset mode to control the output on **P2.4/TA0.1**. When the timer matches **TA0CCR0**=10 the **TA0.1** output is cleared and the **CCIFG** flag in **TA0CCR0** register is set. Each time the **TA0R** matches **TA0CCR1**=7 the **TA0.1** output is toggled and the **CCIFG** flag in **TA0CCR1** register is set. The output is reset when the timer is at maximum, so the first time it matches the timer is counting down. So, the output goes high when the timer matches **TA0CCR1** on the way down, and is cleared when it matches on the way up, see Figure 2.23. The period of the wave will be  $2 * \text{TA0CCR0}$ , and the time it is high will be  $2 * \text{TA0CCR1}$ , therefore the duty cycle will be  $\text{TA0CCR1} / \text{TA0CCR0}$ . Output compare events will again be requested at a rate twice as fast as the resulting square wave frequency. One event is required for the rising edge and another for the falling edge. In the examples below, we make **High** plus **Low** be a constant. By adjusting the ratio of **High** and **Low** the software can control the duty cycle.

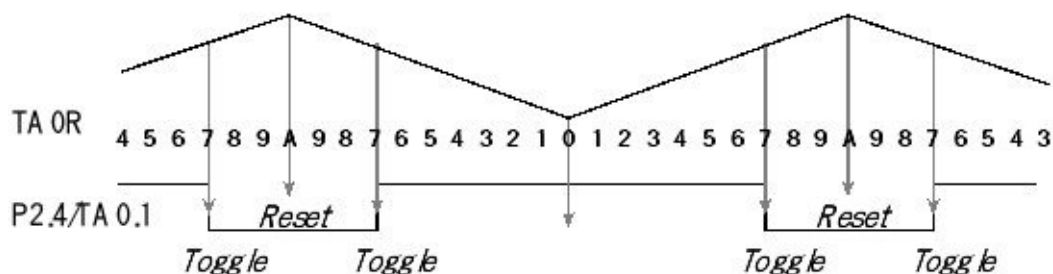


Figure 2.23. The PWM output with timer in up-down mode and output compare in toggle-reset mode.

This implementation occurs in hardware and does not require interrupts. Therefore, it can generate waves close to 0 or 100% duty cycle. Figure 2.24 shows a system using two PWM outputs to control two DC motors. The interface driver will be shown in Section 10.2.

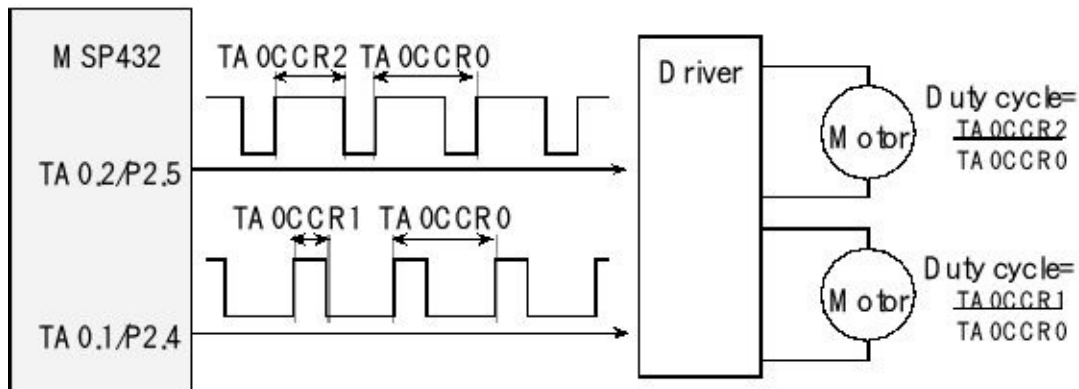


Figure 2.24. The PWM output can adjust the power to two DC motors.

Program 2.19 configures Timer A0 for two PWM outputs. The user calls **PWM\_Init** once to turn it on, and then calls **PWM\_Duty** to adjust the duty cycle.

```

void PWM_Init(uint16_t period, uint16_t duty1, uint16_t duty2){
    Clock_Init48MHz();    // 48 MHz HFXTCLK, SMCLK = 12 MHz
    P2DIR |= 0x30;        // P2.4, P2.5 output
    P2SEL0 |= 0x30;        // P2.4, P2.5 TimerA0 functions
    P2SEL1 &= ~0x30;      // P2.4, P2.5 TimerA0 functions
    TA0CCTL0 = 0x0080;    // CCI0 toggle
    TA0CCR0 = period;     // Period is 2*period*8*83.33ns is 1.333*period
    TA0EX0 = 0x0000;      // divide by 1
    TA0CCTL1 = 0x0040;    // CCR1 toggle/reset
    TA0CCR1 = duty1;      // CCR1 duty cycle is duty1/period
    TA0CCTL2 = 0x0040;    // CCR2 toggle/reset
    TA0CCR2 = duty2;      // CCR2 duty cycle is duty2/period
    TA0CTL = 0x02F0;      // SMCLK=12MHz, divide by 8, up-down mode
}
void PWM_Duty1(uint16_t duty1){
    TA0CCR1 = duty1;      // CCR1 duty cycle is duty1/period
}
void PWM_Duty2(uint16_t duty2){
    TA0CCR2 = duty2;      // CCR2 duty cycle is duty2/period
}

```

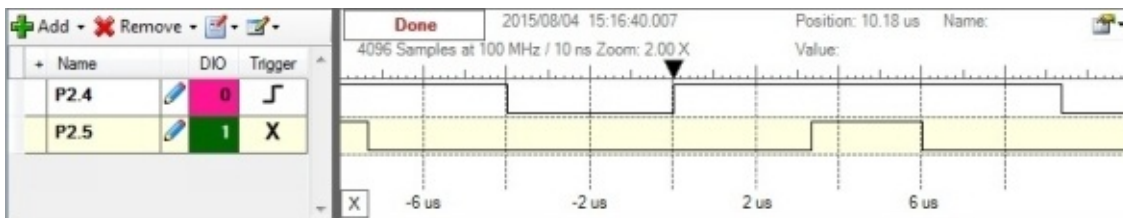
Program 2.19. Software to generate a PWM output using Timer A0

(TimerA0PWM\_MSP432).

**Checkpoint 2.11:** When does an output compare event occur when in PWM mode?

**Checkpoint 2.12:** What happens during an output compare event in PWM mode?

Divide by 8 slows down the 12 MHz SMCLK to count the timer every 666.7ns. Figure 2.25 shows the logic analyzer output when Program 2.19 is called with **PWM\_Init(10,7,2)** creating a 70% duty cycle PWM on P2.4 and a 20% duty cycle PWM on P2.5. Just like Figure 2.11 the timer counts 0 to 10, and then 9 to 1, so there are 20 counts per wave. 20 counts times 666.7ns creates the 13.33µs period for P2.4 and P2.5. When the timer is 7, P2.4 is toggled, and when the timer is 2, P2.5 is toggled.



*Figure 2.25. The PWM output with 13.33µs period and 70% on P2.4 and 20% on P2.5.*

With the counter in up mode, we can use **OUTMOD=7** (reset/set) mode to create PWM outputs. In this mode the period of the wave will be **TA0CCR0+1**, and the time it is high will be **TA0CCR1**, therefore the duty cycle will once again be **TA0CCR1/(TA0CCR0+1)**. When creating PWMs with this approach all outputs will go high at the same time.

## 2.9. Analog Output

A digital to analog convertor (DAC) converts digital signals into analog form as illustrated in Figure 2.26. Although one can interface a DAC to a regular output port, most DACs are interfaced using high-speed synchronous protocols. The DAC output can be current or voltage. Additional analog processing may be required to filter, amplify or modulate the signal. We can also use DACs to design variable gain or variable offset analog circuits.

The DAC **precision** is the number of distinguishable DAC outputs (e.g., 1024 alternatives, 10 bits). The DAC **range** is the maximum and minimum DAC output (volts, amps). The DAC resolution is the smallest distinguishable change in output. The units of resolution are in volts or amps depending on whether the output is voltage or current. The **resolution** is the change in output that occurs when the digital input changes by 1.

$$\text{Range}(\text{volts}) = \text{Precision}(\text{alternatives}) \cdot \text{Resolution}(\text{volts})$$

The DAC **accuracy** is  $(\text{Actual} - \text{Ideal}) / \text{Ideal}$  where Ideal is referred to the National Institute of Standards and Technology (NIST). One can choose the full scale **range** of the DAC to simplify the use of fixed-point math. For example, if an 8-bit DAC had a full scale range of 0 to 2.55 volts, then the resolution would be exactly 10 mV. This means that if the DAC digital input were 12310, then the DAC output voltage would be 1.23 volts.

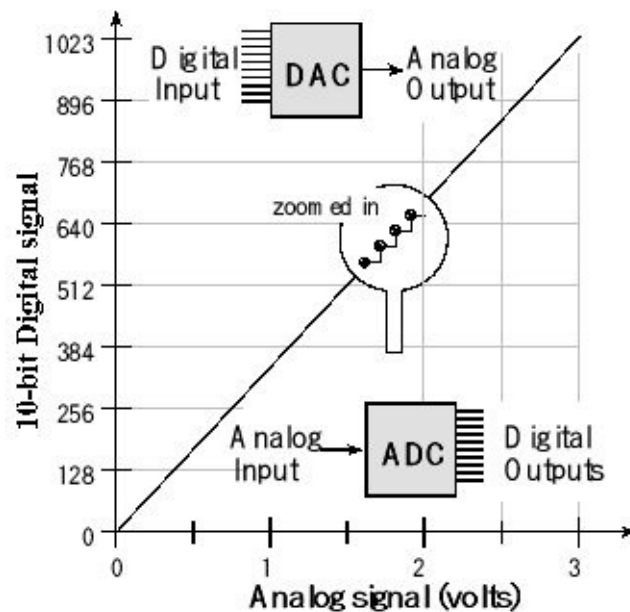


Figure 2.26. A 10-bit DAC provides analog output. A 10-bit ADC provides analog input.

A DAC **gain error** is a shift in the slope of the  $V_{out}$  versus digital input static response. A DAC **offset error** is a shift in the  $V_{out}$  versus digital input static

response. The DAC transient response has three components: delay phase, slewing phase, ringing phase. During the delay phase, the input has changed but the output has not yet begun to change. During the slewing phase, the output changes rapidly. During the ringing phase, the output oscillates while it stabilizes. For purposes of **linearity**, let  $m, n$  be digital inputs, and let  $f(n)$  be the analog output of the DAC, see Figure 2.27. One quantitative measure of linearity is the correlation coefficient of a linear regression fit of the  $f(n)$  responses. If  $\Delta$  is the DAC resolution, it is linear if

$$f(n+1)-f(n) = f(m+1)-f(m) = \Delta \quad \text{for all } n, m$$

The DAC is **monotonic** if

$$\text{sign}(f(n+1)-f(n)) = \text{sign}(f(m+1)-f(m)) \quad \text{for all } n, m$$

Conversely, the DAC is **nonlinear** if

$$f(n+1)-f(n) \neq f(m+1)-f(m) \quad \text{for some } n, m$$

Practically speaking all DACs are nonlinear, but the worst nonlinearity is nonmonotonicity. The DAC is **nonmonotonic** if

$$\text{sign}(f(n+1)-f(n)) \neq \text{sign}(f(m+1)-f(m)) \quad \text{for some } n, m$$

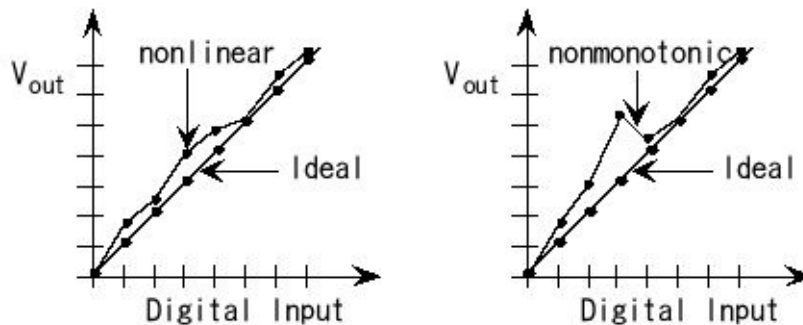
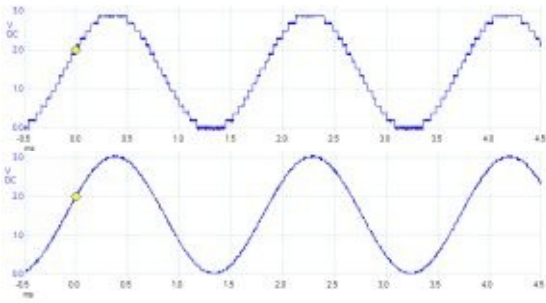


Figure 2.27. Nonlinear and nonmonotonic DACs.

Many manufacturers, like Analog Devices, Texas Instruments, Sipex and Maxim produce DACs. These DACs have a wide range of performance parameters and come in many configurations. The following paragraphs discuss the various issues to consider when selecting a DAC. Although we assume the DAC is used to generate an analog waveform, these considerations will generally apply to most DAC applications.

**Precision/range/resolution.** These three parameters affect the quality of the signal that can be generated by the system. The more bits in the DAC the finer the control the system has over the waveform it creates. As important as this parameter is, it is one of the more difficult specifications to establish a priori. Multiple versions of the software (e.g., 4-bit, 8-bit, 10-bit, and 12-bit DAC) are used to see experimentally the effect of DAC precision on the overall system performance. Figure 2.28 illustrates how DAC precision affects the quality of the generated waveform. DAC parameters of noise include signal to noise ratio (SNR), signal to noise ratio plus distortion (SINAD), and total harmonic distortion (THD)





*Figure 2.28. The waveform on the top uses a 4-bit DAC, while on one on the bottom uses a 12-bit DAC.*

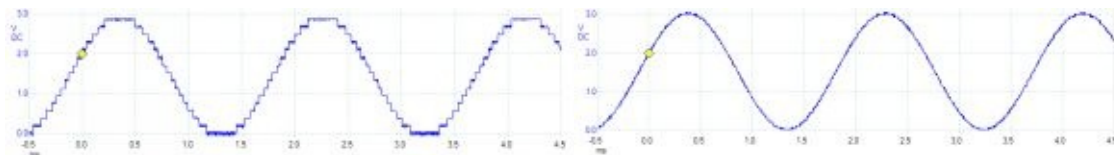
*Channels.* Even though multiple channels could be implemented using multiple DAC chips, it is usually more efficient to design a multiple channel system using a multiple channel DAC. Some advantages of using a DAC with more channels than originally conceived are future expansion, automated calibration, and automated testing. A multiple channel DAC allows you to update all channels at the same time.

*Configuration.* DACs can have voltage or current outputs. Current output DACs can be used in a wide spectrum of applications (e.g., adding gain and filtering), but do require external components. DACs can have internal or external references. An internal reference DAC is easier to use for standard digital input/analog output applications, but the external reference DAC can often be used in variable gain applications (multiplying DAC). Sometimes the DAC generates a unipolar output, while other times the DAC produces bipolar outputs.

*Power.* There are three power issues to consider. The first consideration is the type of power required. Older devices require three power voltages (e.g., +5 and -5 V), while most devices will operate on a single voltage supply (e.g., +2.7, +3.3, or +5 V.) If a single supply can be used to power all the digital and analog components, then the overall system costs will be reduced. The second consideration is the amount of power required. Some devices can operate on less than 0.1 mW and are appropriate for battery-operated systems or for systems where excess heat is a problem. The last consideration is the need for a low-power sleep mode. Some battery operated systems need the DAC only intermittently. In these applications, we wish to give a shutdown command to the DAC, so that it draws less current when not needed.

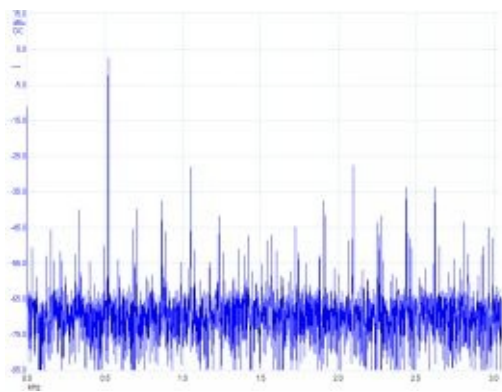
*Speed.* There are a couple of parameters manufacturers use to specify the dynamic behavior of the DAC. The most common is settling time, another is maximum output rate. When operating the DAC in variable gain mode, we are also interested in the gain/bandwidth product of the analog amplifier. When comparing specifications reported by different manufacturers it is important to consider the exact situation used to collect the parameter. In other words, one manufacturer may define settling time as the time to reach 0.1% of the final output after a full scale change in input given a certain load on the output, while another manufacturer may define settling time as the time to reach 1% of the final output after a 1 volt change in input under a different load. The speed of the DAC together with the speed of the computer/software will

determine the effective frequency components in the generated waveforms. Both the software (rate at which the software outputs new values to the DAC) and the DAC speed must be fast enough for the given application. In other words, if the software outputs new values to the DAC at a rate faster than the DAC can respond, then errors will occur. Figure 2.29 illustrates the effect of DAC output rate on the quality of the generated waveform. According to the Nyquist Theorem states the digital data rate must be greater than twice the maximum frequency component of the desired analog waveform. However, both waveforms in Figure 2.29 satisfy the Nyquist Theorem, but increasing the output rate by eight improves the signal to noise ratio by eight. 31 dB is a ratio of about 35 to 1, and 49 dB is a ratio of about 281 to 1. If the goal is to create a sine wave at a fixed frequency, we could improve the SNR greatly by using an analog low pass filter.

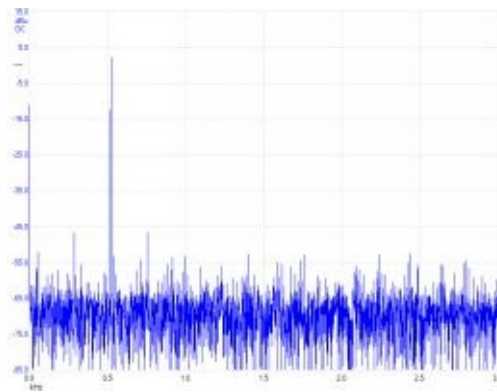


Experimental data of a 32-output 523 Hz sine-wave sine-wave

Experimental data of a 256-output 523 Hz sine-wave



Signal/noise ratio is 31 dB (3dB- -28dB)



Signal/noise ratio is 49 dB (3dB- -46dB)

*Figure 2.29. The waveform on the right was created by a system with eight times the output rate than the left. Voltage versus time data on top and the Fourier Transform (frequency spectrum dB versus kHz) of the data on the bottom. There is a point in the spectrum at 0, which is the DC component. However, the signal is the 523 Hz bump with a magnitude of 3dB, representing the sine wave. The noise are all the other points not at 0 or 523 Hz. The largest noise on the left is -28 dB. The largest noise on the right is -46 dB.*

*Interface.* Three approaches exist for interfacing the DAC to the computer. In a digital logic or parallel interface, the individual data bits are connected to a dedicated computer output port. For example, a 12-bit DAC requires a 12-bit output port bits to interface. The software simply writes to the parallel port(s) to change the DAC output. The second approach is called  $\mu$ P-bus or microprocessor-compatible.

These devices are intended to be interfaced onto the address/data bus of an expanded mode microcontroller. The third approach is a high-speed serial interface like I<sup>2</sup>C or SPI. This approach requires the fewest number of I/O pins. Even if the microcontroller does not support the SPI interface directly, these devices can be interfaced to regular I/O pins via the bit-banging software approach.

*Package.* DIP packages are convenient for creating and testing an original prototype. On the other hand, surface mount packages require less board space. Because surface mount packages do not require holes in the PC board, circuits with these devices are easier/cheaper to produce.

*Cost.* Cost is always a factor in engineering design. Beside the direct costs of the individual components in the DAC interface, other considerations that affect cost include: 1) power supply requirements; 2) manufacturing costs; 3) the labor involved in individual calibration if required; and 4) software development costs.

## 2.10. Analog Input

### 2.10.1. ADC Parameters

An analog to digital converter (ADC) converts an analog signal into digital form. The input signal is usually an analog voltage ( $V_{in}$ ), and the output is a binary number. The ADC **precision** is the number of distinguishable ADC inputs (e.g., 4096 alternatives, 12 bits). The ADC **range** is the maximum and minimum ADC input (volts, amps). The ADC **resolution** is the smallest distinguishable change in input (volts, amps). The resolution is the change in input that causes the digital output to change by 1.

$$\text{Range}(\text{volts}) = \text{Precision}(\text{alternatives}) \cdot \text{Resolution}(\text{volts})$$

Normally we don't specify accuracy for just the ADC, but rather we give the accuracy of the entire system (including transducer, analog circuit, ADC and software). Therefore, accuracy is defined as part of the systems approach to data acquisition systems. An ADC is **monotonic** if it has no missing codes. This means if the analog signal is a slow rising voltage, then the digital output will hit all values sequentially. The ADC is linear if the resolution is constant through the range. Let  $f(x)$  be the input/output ADC transfer function. One quantitative measure of **linearity** is the correlation coefficient of a linear regression fit of the  $f(x)$  responses. The ADC **speed** is the time to convert, called  $t_c$ . The ADC **cost** is a function of the number and price of internal components. There are four common encoding schemes for an ADC. Table 2.23 shows two encoding schemes for a 12-bit unipolar ADC.

Unipolar Codes	Straight Binary	Complementary Binary
$+V_{max}$	1111,1111,1111	0000,0000,0000
$+V_{max}/2$	1000,0000,0000	0001,1111,1111
$+V_{max}/1024$	0000,0000,0001	1111,1111,1110
+0.00	0000,0000,0000	1111,1111,1111

**Table 2.23. Unipolar codes for a 12-bit ADC with a range of 0 to  $+V_{max}$ .**

The ADCs on the MSP432 (14 bits) and TM4C (12 bits) families use straight binary. The MSP432 has a range of 0 to 2.5V, and the TM4C has a range of 0 to 3.3 V. To convert between straight binary and complementary binary we simply complement (change 0 to 1, change 1 to 0) all the bits. To convert between offset binary and 2's complement, we complement just the most significant bit. The exclusive-or operation can be used to complement bits.

Just like the DAC, one can choose the full scale range to simplify the use of fixed-

point math. For example, if a 10-bit ADC had a full scale range of 0 to 1.023 volts, then the resolution would be exactly 1 mV. This means that if the ADC input voltage were 0.234 volts, then the result would be  $234_{10}$ .

The **total harmonic distortion** (THD) of a signal is a measure of the harmonic distortion present and is defined as the ratio of the sum of the powers of all harmonic components to the power of the fundamental frequency. Basically, it is a measure of all the noise processes in an ADC and usually is given in dB full scale. A similar parameter is **signal-to-noise and distortion ratio** (SINAD), which is measured by placing a pure sine wave at the input of the ADC (signal) and measuring the ADC output (signal plus noise). We can compare precision in bits to signal-to-noise ratio in dB using the relation  $\text{dB} = 20 \log_{10}(2^n)$ . For example, the 12-bit MAX1247 ADC has a SINAD of 73 dB. Notice that  $20 \log_{10}(2^{12})$  is 72 dB. The ADCs on most microcontrollers use the **successive approximation** technique.

For a discussion of ADC techniques, see Chapter 8 of Volume 2.

## 2.10.2. Internal ADC on TM4C

Table 2.24 shows the ADC register bits required to perform periodic sampling on a single channel. For more complex configurations refer to the specific data sheet. The TM4C123 and TM4C1294 can sample up to 1 million samples per second, see Table 2.25. Running the ADC slower will make it more accurate, and use less power.

Address	31-2				1	0	Name		
\$400F.E638					ADC1	ADC0	SYSCTL_RCGCADC_R		
	31-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0	
\$4003.8020		SS3		SS2		SS1		SS0	ADC0_SSPRI_R
	31-16			15-12	11-8	7-4	3-0		
\$4003.8014				EM3	EM2	EM1	EM0		ADC0_EMUX_R
	31-4			3	2	1	0		
\$4003.8000				ASEN3	ASEN2	ASEN1	ASEN0		ADC0_ACTSS_R
\$4003.8028				SS3	SS2	SS1	SS0		ADC0_PSSI_R
\$4003.8004				INR3	INR2	INR1	INR0		ADC0_RIS_R
\$4003.8008				MASK3	MASK2	MASK1	MASK0		ADC0_IM_R
\$4003.8FC4				Speed					ADC0_PC_R
\$4003.800C				IN3	IN2	IN1	IN0		ADC0_ISC_R
	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0	
\$4003.8040	MUX7	MUX6	MUX5	MUX4	MUX3	MUX2	MUX1	MUX0	ADC0_SSMUX0_R
	31-16	15-12	11-8	7-4	3-0				
\$4003.8060		MUX3	MUX2	MUX1	MUX0				ADC0_SSMUX1_R
\$4003.8080		MUX3	MUX2	MUX1	MUX0				ADC0_SSMUX2_R
\$4003.80A0									ADC0_SSMUX3_R

	31	30	29	28	27	26	...	8	7	6	5	4	3	2	1	0	
\$4003.8044	TS7	IE7	END7	D7	TS6	IE6	...	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0	ADC0_SSCTL0_R
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
\$4003.8064	TS3	IE3	END3	D3	TS2	IE2	END2	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0	ADC0_SSCTL1_R
\$4003.8084	TS3	IE3	END3	D3	TS2	IE2	END2	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0	ADC0_SSCTL2_R
\$4003.80A4													TS0	IE0	END0	D0	ADC0_SSCTL3_R
	31-10							11-0									
\$4003.8048								DATA							ADC0_SSFIFO0_R		
\$4003.8068								DATA							ADC0_SSFIFO1_R		
\$4003.8088								DATA							ADC0_SSFIFO2_R		
\$4003.80A8								DATA							ADC0_SSFIFO3_R		

**Table 2.24. Some of the ADC registers. Each register is 32 bits wide.**

The ADC has four sequencers, but we will use only sequencer 3. We set the **ADC\_SSPRI\_R** register to 0x3210 to make sequencer 3 the lowest priority. Because we are using just one sequencer, we just need to make sure each sequencer has a unique priority. We set bits 15–12 (**EM3**) in the **ADC\_EMUX\_R** register to specify how the ADC will be triggered. Table 2.26 shows the various ways to trigger an ADC conversion. In this section we will use timer triggering (**EM3**=0x5). If we specify software start (**EM3**=0x0), then the software writes an 8 (**SS3**) to the **ADC\_PSSI\_R** to initiate a conversion on sequencer 3. Bit 3 (**INR3**) in the **ADC\_RIS\_R** register will be set when the conversion is complete.

We can enable and disable the sequencers using the **ADC\_ACTSS\_R** register. There are four sequencers on the TM4C123. Which channel we sample is configured by writing to the **ADC\_SSMUX3\_R** register. The **ADC\_SSCTL3\_R** register specifies the mode of the ADC sample. We set **TS0** to measure temperature and clear it to measure the analog voltage on the ADC input pin. We set **IE0** so that the **INR3** bit is set on ADC conversion, and clear it when no flags are needed. We will set **IE0** for both interrupt and busy-wait synchronization. When using sequencer 3, there is only one sample, so **END0** will always be set, signifying this sample is the end of the sequence. We set the **D0** bit to activate differential sampling, such as measuring the analog difference between ADC1 and ADC0 pins. In our example, we clear **D0** to sample a single-ended analog input. The **ADC\_RIS\_R** register has flags that are set when the conversion is complete, assuming the **IE0** bit is set. The **ADC\_IM\_R** register has interrupt arm bits. The **ADC\_ISC\_R** register has interrupt trigger bits. The **IN3** bit is set when both **INR3** and **MASK3** are set. We clear the **INR3** and **IN3** bits by writing an 8 to the **ADC\_ISC\_R** register. The interrupt vector for ADC sequencer 3 is at 0x00000084.

<i>Value</i>	<i>Description</i>
0x7	1M samples/second

0x5	500K samples/second
0x3	250K samples/second
0x1	125K samples/second

**Table 2.25. The Speed bits in the ADC0\_PC\_R register.**

<i>Value</i>	<i>Event</i>
0x0	Software start
0x1	Analog Comparator 0
0x2	Analog Comparator 1
0x3, 0x9-0x0E	Reserved
0x4	External (GPIO PB4)
0x5	Timer
0x6	PWM0
0x7	PWM1
0x8	PWM2
0xF	Always (continuously sample)

**Table 2.26. The ADC EM3, EM2, EM1, and EM0 bits in the ADC\_EMUX\_R register.**

There are 13 steps to configure the ADC to sample a single channel at a periodic rate. The most accurate sampling method is timer-triggered sampling (**EM3=0x5**). On the TM4C123, the MUX fields are 4 bits wide, allowing us to specify channels 0 to 11. On the TM4C1294, the channel ranges from 0 to 19. See Tables 1.4 and 1.5 to see mapping from pin to channel.

**Step 1.** We enable the ADC clock in the **SYSCTL\_RCGCADC\_R** register.

**Step 2.** Bits 3 – 0 of the **ADC0\_PC\_R** register specify the maximum sampling rate of the ADC. In this example, we will sample slower than 125 kHz, so the maximum sampling rate is set at 125 kHz. This will require less power and produce a longer sampling time as described the S/H section, creating a more accurate conversion.

**Step 3.** We will set the priority of each of the four sequencers. In this case, we are using just one sequencer, so the priorities are irrelevant, except for the fact that no two sequencers should have the same priority. The default configuration has Sample Sequencer 0 with the highest priority, and Sample Sequencer 3 as the lowest priority.

**Step 4.** Next, we need to configure the timer to run at the desired sampling frequency. We enable the Timer0 clock by setting bit 0 of the **SYSCTL\_RCGCTIMER\_R** register. This initialization is similar to Program 2.6 with two changes. First we set bit 5 of the **TIMER0\_CTL\_R** register to activate **TAOTE**, which is the Timer A output trigger enable. Secondly, we do not arm any Timer0 interrupts. The rate at which the timer rolls over determines the sampling frequency. Let *prescale* be the

value loaded into **TIMER0\_TAPR\_R**, and let *period* be the value loaded into **TIMER0\_TAILR\_R**. If the period of the bus clock frequency is  $\Delta t$ , then the ADC sampling period will be

$$\Delta t * (\textit{prescale} + 1) * (\textit{period} + 1)$$

The fastest sampling rate is determined by the speed of the processor handling the ADC interrupts and by the speed of the main program consuming the data from the FIFO. If the bus clock is 80 MHz, the slowest possible sampling rate for this example is  $80\text{MHz}/2^{32}$ , which is about 0.018 Hz, which is every 53 seconds.

**Step 5.** Before configuring the sequencer, we need to disable it. To disable sequencer 3, we write a 0 to bit 3 (**ASEN3**) in the **ADC0\_ACTSS\_R** register. Disabling the sequencer during programming prevents erroneous execution if a trigger event were to occur during the configuration process.

**Step 6.** We configure the trigger event for the sample sequencer in the **ADC0\_EMUX\_R** register. For this example, we write a 0101 to bits 15–12 (**EM3**) specifying timer trigger mode.

**Step 7.** For each sample in the sample sequence, configure the corresponding input source in the **ADC0\_SSMUXn** register. In this example, we write the channel number (0, 1, 2, or 3) to bits 3–0 in the **ADC0\_SSMUX3\_R** register.

**Step 8.** For each sample in the sample sequence, we configure the sample control bits in the corresponding nibble in the **ADC0\_SSCTLn** register. When programming the last nibble, ensure that the **END** bit is set. Failure to set the **END** bit causes unpredictable behavior. Sequencer 3 has only one sample, so we write a 0110 to the **ADC0\_SSCTL3\_R** register. Bit 3 is the **TS0** bit, which we clear because we are not measuring temperature. Bit 2 is the **IE0** bit, which we set because we want to request an interrupt when the sample is complete. Bit 1 is the **END0** bit, which is set because this is the last (and only) sample in the sequence. Bit 0 is the **D0** bit, which we clear because we do not wish to use differential mode.

**Step 9.** If interrupts are to be used, write a 1 to the corresponding mask bit in the **ADC0\_IM\_R** register. We want an interrupt to occur when the conversion is complete (set bit 3, **MASK3**).

**Step 10.** We enable the sample sequencer logic by writing a 1 to the corresponding **ASENn**. To enable sequencer 3, we write a 1 to bit 3 (**ASEN3**) in the **ADC0\_ACTSS\_R** register.

**Step 11.** The priority of the ADC0 sequencer 3 interrupts are in bits 13–15 of the **NVIC\_PRI4\_R** register.

**Step 12.** Since we are requesting interrupts, we need to enable interrupts in the NVIC. ADC sequencer 3 interrupts are enabled by setting bit 17 in the **NVIC\_EN0\_R** register.

**Step 13.** Lastly, we must enable interrupts in the **PRIMASK** register.



The timer starts the conversion at a regular rate. Bit 3 (**INR3**) in the **ADC0\_RIS\_R** register will be set when the conversion is done. This bit is armed and enabled for interrupting, so conversion complete will trigger an interrupt. The **IN3** bit in the **ADC0\_ISC\_R** register triggers the interrupt. The ISR acknowledges the interrupt by writing a 1 to bit 3 (**IN3**). The 12-bit result is read from the **ADC0\_SSFIFO3\_R** register. The book web site for has example code. In order to reduce latency of other interrupt requests in the system, this ISR simply stores the 12-bit conversion in a FIFO, to be processed later in the main program. Program 2.20 shows the initialization and interrupt service routine to affect the periodic sampling. For the port pin, we disable its DEN, clear its DIR, set its AFSEL and enable its AMSEL bit.

```

void ADC0_InitTimer0ATriggerSeq3PD3(uint32_t period){
    volatile uint32_t delay;
    SYSCTL_RCGCADC_R |= 0x01;    // 1) activate ADC0
    SYSCTL_RCGCGPIO_R |= 0x08;   // Port D clock
    delay = SYSCTL_RCGCGPIO_R;   // allow time for clock to stabilize
    GPIO_PORTD_DIR_R &= ~0x08;   // make PD3 input
    GPIO_PORTD_AFSEL_R |= 0x08;  // enable alternate function on PD3
    GPIO_PORTD_DEN_R &= ~0x08;   // disable digital I/O on PD3
    GPIO_PORTD_AMSEL_R |= 0x08;  // enable analog functionality on PD3
    ADC0_PC_R = 0x01;            // 2) configure for 125K samples/sec
    ADC0_SSPRI_R = 0x3210;       // 3) seq 0 is highest, seq 3 is lowest
    SYSCTL_RCGCTIMER_R |= 0x01; // 4) activate timer0
    delay = SYSCTL_RCGCGPIO_R;
    TIMER0_CTL_R = 0x00000000;   // disable timer0A during setup
    TIMER0_CTL_R |= 0x00000020;  // enable timer0A trigger to ADC
    TIMER0_CFG_R = 0;            // configure for 32-bit timer mode
    TIMER0_TAMR_R = 0x00000002;  // configure for periodic mode
    TIMER0_TAPR_R = 0;           // prescale value for trigger
    TIMER0_TAILR_R = period-1;   // start value for trigger
    TIMER0_IMR_R = 0x00000000;   // disable all interrupts
    TIMER0_CTL_R |= 0x00000001;  // enable timer0A 32-b, periodic
    ADC0_ACTSS_R &= ~0x08;       // 5) disable sample sequencer 3
    ADC0_EMUX_R = (ADC0_EMUX_R&0xFFFF0FFF)+0x5000; // 6) timer trigger
    ADC0_SSMUX3_R = 4;           // 7) PD3 is analog channel 4
    ADC0_SSCTL3_R = 0x06;        // 8) set flag and end after first sample
    ADC0_IM_R |= 0x08;           // 9) enable SS3 interrupts
    ADC0_ACTSS_R |= 0x08;        // 10) enable sample sequencer 3
    NVIC_PRI4_R = (NVIC_PRI4_R&0xFFFF00FF)|0x00004000; // 11)priority 2
    NVIC_EN0_R = 1<<17;         // 12) enable interrupt 17 in NVIC
    EnableInterrupts();          // 13) enable interrupts
}

void ADC0Seq3_Handler(void){
    ADC0_ISC_R = 0x08;           // acknowledge ADC sequence 3 completion
    Fifo_Put(ADC0_SSFIFO3_R);    // pass to foreground
}

```

}

*Program 2.20. Software to sample data using the ADC (ADCT0ATrigger\_xxx).*

The above example only samples one analog input. The **ADCSWTriggerTwoChan\_xxx** project samples two channels using software start.

### 2.10.3. Internal ADC on MSP432

Table 2.27 shows the ADC register bits required to perform sampling on a single channel. For more complex configurations refer to the specific data sheet. When converting from analog to digital we can select speed (how fast it runs), power (how much energy it takes) and accuracy (the number of bits in the result). For example, to reduce power we can run slower or reduce the number of bits. Bits 4 – 0 in **ADC14MCTL0** specify the channel to convert. See Table 2.3 to see the mapping between I/O pins and the ADC analog input channel. For example, channel 6 exists on pin P4.7. On the MSP432, we will need to set bits in the **SEL0 SEL1** bits to 11 to activate the analog interface. Most of the ADC control bits can only be set when **ADC14ENC = 0**, so clearing this bit will occur first during initialization.

	31-30	29-27	26	25	24-22	21-19	18-17	16		
0x40012000	PDIV	SHSx	SHP	ISSH	DIVx	SSELx	CONsx	BUSY	ADC14CTL0	
	15-12	11-8	7	6-5	4	3-2	1	0		
	SHT1x	SHT0x	MSC		ON		ENC	SC	ADC14CTL0	
	31-28	27 – 24				22	21	20-16		
0x40012004		CH3MAP – CH0MAP				BATmap		CStartAdr	ADC14CTL1	
	15-6		5 – 4		3	2		1-0		
			RES		DF	REFBURST		PWRMD	ADC14CTL1	
	31-16	15		14	13	12	11-8			
0x40012018		WINCTH		WINC	DIF		VRSEL		ADC14MCTL0	
	7	6	5	4 – 0						
	EOS			ADC14INCHx					ADC14MCTL0	
	31 – 16				15 – 0					
0x40012098					Conversion_Results					ADC14MEM0
	31		5	4	3	2	1	0		
0x4001213C	IE31	...	IE5	IE4	IE3	IE2	IE1	IE0	ADC14IER0	
	31		5	4	3	2	1	0		
0x40012144	IFG31	...	IFG5	IFG4	IFG3	IFG2	IFG1	IFG0	ADC14IFGR0	

**Table 2.27. The MSP432 ADC registers. Each register is 32 bits wide.**

The **PDIV** field selects a ADC clock divider (00 is divide by 1, 01 is divide by 4, 10 is divide 32, and 11 is divide by 64). Running with a slower clock increases

accuracy but will take longer to convert. We will set the **SHSx** field to 000 to select the **ADC14SC** signal as the sample and hold source. **SHP** is the sample and hold pulse mode select. With **SHP=0** the ADC runs faster. The **ISSH** bit can be used to invert the sample and hold pulse. We will clear this bit. We use the 3-bit **DIVx** field to select another ADC clock divider. If the value of this field is  $n$ , then there will be a divide by  $n+1$ . Again this defines a tradeoff between accuracy and speed. The 3-bit field **SSELx** defines the clock source. We will set it to 100 to select the SMCLK. For other choices see Table 2.28.

<i>Value</i>	<i>ADC Clock Source</i>
000	MODCLK
001	SYSCLK
010	ACLK
011	MCLK
100	SMCLK
101	HSMCLK

**Table 2.28. The ADC clock selection SSELx bits.**

The ADC has a sample and hold module (SHM) at its input. The first ADC conversion step is to put the SHM in sample mode during which time the analog signal is connected to a sampling capacitor. Current flows as the voltage on the capacitor rises or falls to equalize to the analog input voltage. The second step is to disconnect the capacitor from the analog input, hold mode. The ADC converts the voltage on the capacitor to digital form. The longer the sampling phase, the more accurate will be the conversion. The **SHT1x** and **SHT0x** are 4-bit fields defining the length of the sampling period. **SHT0x** controls registers **ADC14MEM0** to **ADC14MEM7** and **ADC14MEM24** to **ADC14MEM31**. Since we will be using **ADC14MEM0**, we set **SHT0x**. Table 2.29 lists the sampling periods available.

<i>Value</i>	<i>Sampling Period</i>
0000	4 ADC14CLK periods
0001	8 ADC14CLK periods
0010	16 ADC14CLK periods
0011	32 ADC14CLK periods
0100	64 ADC14CLK periods
0101	96 ADC14CLK periods
0110	128 ADC14CLK periods
0111	192 ADC14CLK periods

**Table 2.29. The SHT0x SHT1x fields define the sampling period.**

The **MSC** bit selects single or multiple conversions. We will clear this bit so when

the software starts conversion it takes sample and stops. We set the **ON** bit to apply power to the ADC. We set the **ENC** bit to enable the ADC. As mentioned earlier we clear the **ENC** bit while configuring the ADC. The software will set the **SC** bit to start an ADC conversion. Software writes one to **SC** but this bit is automatically cleared.

There are 32 **ADC14MEMx** registers,  $x = 0 - 31$ , similar to **ADC14MEM0** and 32 **ADC14MCTLx** registers similar to **ADC14MCTL0** shown in Table 2.27. The 5-bit **CStartAdr** field specifies the conversion start address. These bits select which ADC14 conversion memory register is used for a single conversion or for the first conversion in a sequence. The value of **CStartAdr** is 0 to 31, corresponding to **ADC14MEM0** to **ADC14MEM31**. We will use **ADC14MEM0** and **ADC14MCTL0** in our example by setting **CStartAdr** to 0.

The **RES** field specifies the ADC resolution. Again we can trade off accuracy for speed. Set **RES** to 00 for 8-bit conversion, set **RES** to 01 for 10 bits, set **RES** to 10 for 12 bits and set it to 11 for 14 bits. We set the **REFBURST** bit if we desire to turn off the reference when not in use. In our example, we will clear this bit to have the reference on continuously.

The **PWRMD** field defines the power modes. Setting it to 00 will use the most power but allow for 14-bit conversions at the highest speed. We set **PWRMD** to 10 for low-power mode and can be used for 12-bit, 10-bit and 8-bit resolutions.

We perform the following steps to timer-trigger the ADC and sample data periodically using interrupt synchronization, see Program 2.21. This method has no sampling jitter.

**Step 1.** Halt the timer during initialization

**Step 2.** We enable the timer to use **SMCLK**, divide by 1, stop mode, and disable interrupts. Interrupts will be generated by the ADC module when the conversion is complete and not by the timer when the conversion is started.

**Step 3.** We configure the timer to start the ADC conversion periodically. In particular, bits 15-10 are 0 because we do not need capture events. Bit 8 is zero to use compare mode. Bits 7-5 are 011 to create set/reset output mode, which will be a squarewave created automatically by the timer and sent to the ADC. The frequency of this squarewave will set the ADC sampling rate. An analog-to-digital conversion is initiated with a rising edge of the timer squarewave output. Bit 4 is clear because the timer does not create interrupts.

**Step 4.** In this step we set the sampling period. If the **SMCLK** is 12 MHz, then 1 ms period output will be created if we write a 5999 into **TA0CCR1** and we write a 11999 into **TA0CCR0**.

**Step 5.** This step configures the timer clock as divide by 1.

**Step 6.** Before configuring the analog reference, we make sure it is idle.

**Step 7.** Bits 5-4 (**REFVSEL**) set to 1,1 to select the 2.5V reference. This defines the ADC range to be 0 to 2.5V. Bit 3 (**REFTCOFF**) is set to disable the temperature sensor. Disabling the sensor saves power. Bit 1 (**REFOUT**) is clear to disconnect

the reference from P5.6 .Bit 0 (**REFON**) is set to enable the reference.

**Step 8.** After configuring the analog reference, we wait for it to stabilize.

**Step 9.** Before configuring the ADC, we disable it. Clearing bit 1 (**ADC14ENC**) allows us to program the ADC modes.

**Step 10.** Before configuring the ADC, we make sure it is idle.

**Step 11.** We write to the **ADC14CTL0** register to set the ADC conversion mode. Bits 31-30 (**PDIV**) are set to 0,0 to specify a predivide by 1. Bits 29-27 (**SHSx**) are set to 0,0,1 to select TA0\_C1 output as the ADC trigger source. Again, a rising edge of the timer output will initiate an ADC conversion. We set bit 26 (**SHP**) to make the sample/hold use pulse mode. We clear bit 25 (**ISSH**) so the sample-and-hold is not inverted. We set bits 24-22 (**DIVx**) to 0,0,0 to the clock divider to 1. We set bits 21-19 (**SSELx**) to 1,0,0 to select the **SMCLK** to run the ADC. We set bits 18-17 (**CONSEQx**) to 1,0 to set the ADC mode to *Repeat-single-channel*. We will set both bits 15-12 (**SHT1x**) and bits 11-8 (**SHT0x**) to select 32 clocks each for sample-and-hold times 1 and 0. The longer we sample the more accurate the result, but the longer it takes to do the conversion. We clear bit 7 (**MSC**) so there is one sample per rising edge of the trigger. Set bit 4 (**ON**) to power up the ADC.

**Step 12.** We write to the **ADC14CTL1** register to set additional ADC modes. We set bits 20-16 (**STARTADDx**) to 0,0,0,0,0 to use **ADC14MEM0** as the starting address. We set bits 5-4 (**RES**) to 1,1 to select 14-bit conversion requiring 16 clocks. Clearing bit 3 (**DF**) specifies binary unsigned mode. Clearing bit 2 (**REFBURST**) will power the reference continuously. Clearing bits 1-0 (**PWRMD**) specifies regular power mode. It takes more power to leave the power on, but the results will be more accurate.

**Step 13.** Writing to the **ADC14MCTL0** register the range and the channel. We clear bit 14 (**WINC**) to disable the comparator. We clear bit 13 (**DIF**) to specify single-ended mode. We set bits 11-8 (**VRSEL**) to 0,0,0,1 to set the positive reference to **VREF** (2.5V) and the negative reference to ground. We set bit 7 (**EOS**) to activate an end of sequence event. Bits 4-0 (**INCHx**) set the input channel. Writing a 6 specifies channel 6, which is P4.7.

**Step 14.** In this step we arm the IFG0 for interrupts and disarm the other flags.

**Step 15.** We set the **SEL0** and **SEL1** bits for P4.7 to specify analog input.

**Step 16.** we set the **ENC** bit to enable the ADC.

**Step 17.** We specify the priority of the ADC interrupt. Because the trigger occurs in hardware this interrupt priority needs to high enough so the ISR is run within 1 ms (before another sample would be triggered).

**Step 18.** We enable ADC interrupts in the NVIC

**Step 19.** Lastly, we activate the timer to begin sampling. Interrupts will be enabled in the main program after all devices initialized

```
void ADC0_InitTA0TriggerCh6(uint16_t period){
    TA0CTL &= ~0x0030; // 1) halt Timer A0
    TA0CTL = 0x0200; // 2)SMCLK, stop mode, divide by one, no interrupt
    TA0CCTL1 = 0x0060; // 3) no capture, compare mode, set/reset
    TA0CCR1 = (period-1)/2; // 4) specify sampling period
```

```

TA0CCR0 = (period - 1);
TA0EX0 &= ~0x0007; // 5) configure for input clock divider /1
while(REFCTL0&0x0400){}; // 6) wait for the reference to be idle
REFCTL0 = 0x0039; // 7) configure reference for static 2.5V
while((REFCTL0&0x1000) == 0){}; // 8) wait for reference to stabilize
ADC14CTL0 &= ~0x00000002; // 9) allow programming
while(ADC14CTL0&0x00010000){}; // 10) wait for BUSY to be zero
ADC14CTL0 = 0x0C243310; // 11) ADC mode
ADC14CTL1 = 0x00000030; // 12) ADC14MEM0, 14-bit, ref on, regular
ADC14MCTL0 = 0x00000186; // 13) 0 to 2.5V, channel 6
ADC14IER0 = 0x00000001; // 14) enable ADC14IFG0 interrupt
ADC14IER1 = 0; // disable these interrupts
P4SEL1 |= 0x80; // 15) analog mode on A6, P4.7
P4SEL0 |= 0x80;
ADC14CTL0 |= 0x00000002; // 16) enable
NVIC_IPR6 = (NVIC_IPR6&0xFFFFF00)|0x00000040; // 17) priority 2
NVIC_ISER0 = 0x01000000; // 18) enable interrupt 24 in NVIC
TA0CTL |= 0x0014; // 19) reset and start Timer A0 in up mode
}
void ADC14_IRQHandler(void){ uint16_t result;
if((ADC14IFGR0&0x00000001) == 0x00000001){
    Fifo_Put(ADC14MEM0);} // pass to foreground
}

```

*Program 2.21. Software to sample data using the ADC (ADC\_TA0Trigger\_MSP432).*

**Checkpoint 2.13:** If the input voltage is 1.0V, what value, in 14-bit unsigned binary mode, will the MSP432 ADC return (assuming 0 to 2.5V range)? What will a TM4C with a 12-bit ADC return (assuming 0 to 3.3V range)?

The above example only samples one analog input. The `ADCSWTriggerTwoChan_MSP432` project samples two channels using software start.

## 2.10.4. IR distance measurement

A **nonmonotonic** response is an input/output function that does not have a mathematical inverse. For example, if two or more input values yield the same output value, then the transducer is nonmonotonic. Software will have a difficult time correcting a nonmonotonic transducer. For example, the Sharp GP2Y0A21YK IR distance sensor has a transfer function as shown in Figure 2.30. If you read a transducer voltage of 2 V, you cannot tell if the object is 3 cm away or 12 cm away.

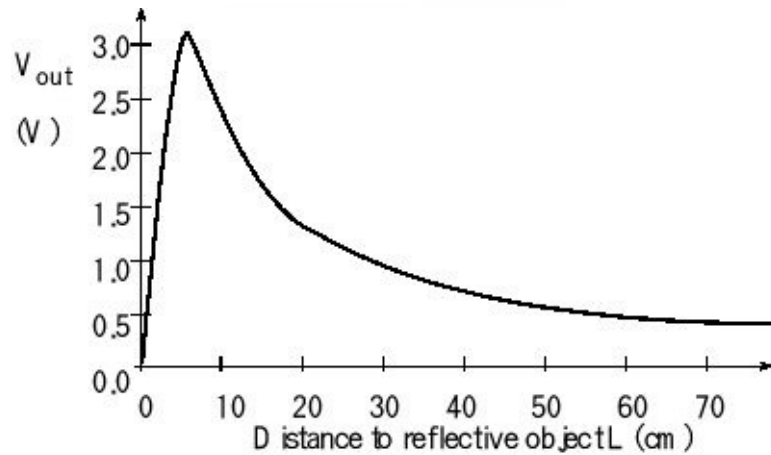


Figure 2.30. The Sharp IR distance sensor exhibits nonmonotonic behavior.

The transducer in Figure 2.17 uses IR light to measure distance to a reflecting object. These sensors require a nonuniform power, so placing a 10  $\mu\text{F}$  near the power line of the sensor reduces noise on other components. If the object is more than 6 cm away, the output voltage is inversely related to distance. If  $N$  is the ADC sample, then distance can be calculated as

$$d = c/N \quad \text{where } c \text{ is a calibration constant}$$

Figure 2.31 shows this sensor has a significant amount of noise. The nonlinear median filter, presented in Chapter 6, is a good choice to improve signal to noise ratio.



Figure 2.31. Noise on a GP2Y0A21YK IR distance sensor shows large periodic spikes.

---

## 2.11. OS Considerations for I/O Devices

### 2.11.1 Board Support Package

The entire book deals with interfacing I/O devices to build embedded systems. However, in this section we will study two considerations of how the OS can manage I/O. It is good design practice to provide an abstraction for the I/O layer. Names for this abstraction include hardware abstraction layer (HAL), device driver, and board support package (BSP). From an operating system perspective, the goal is to make it easier to port the system from one hardware platform to another. The system becomes more portable if we create a BSP for our hardware devices. A BSP could allow you to encapsulate the following:

- Timer initialization
- ISR Handlers
- LED output functions
- Switch input functions
- Setting up the interrupt controller
- Setting up communication channel
- CAN, I2C, ADC, DAC, SPI, serial, graphics

---

**Example 2.1.** Design a BSP for using a periodic interrupt.

**Solution:** In any abstraction, we need to separate what the system does from how it does it. What we use a periodic interrupt for is to run a task at a fixed rate. How we do it on the microcontroller is to enable the SysTick timer and configure it to interrupt periodically, as presented previously in Section 2.2.2. What the user needs is an OS function that he or she can call specifying their task and how often it should run.

We can abstract the periodic interrupt, by defining the function in Program 2.22, which is essentially Program 2.5 with the flexibility to specify the task to run and the period with which to run it. We have hidden from the user the details of the microcontroller. To run the function **Task** once a second, the user calls **OS\_AddPeriodicTask(1000,&Task);**

```
uint32_t static volatile Count;  
uint32_t static Period;  
void (*CallBack)(void); // call back function  
void SysTick_Handler(void){
```



```

Count++;
if(Count==Period){
    Count = 0;
    (*CallBack)();    // execute call back process
}
}
//----- OS_AddPeriodicTask -----
// Input: thePeriod is a time period in ms
//      fp is a function to be executed at this period
// Output: none
// Example: to toggle PD0 once a second, we can
// void toggle(void){PORTD0 ^= 0x01;}
// OS_AddPeriodicTask(1000,&toggle);
void OS_AddPeriodicTask(uint32_t thePeriod, void(*fp)(void)){
    DisableInterrupt();    // make initialization ritual atomic
    Period = thePeriod;
    CallBack = fp;
    Count = 0;
    NVIC_ST_CTRL_R = 0;    // disable SysTick during setup
    NVIC_ST_RELOAD_R = 49999; // reload value, 1ms
    NVIC_ST_CURRENT_R = 0;    // any write to current clears it
    NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R&0x00FFFFFF)|0x40000000; //priority
2
    NVIC_ST_CTRL_R = 0x00000007;// enable with core clock and interrupts
    EnableInterrupts();
}

```

*Program 2.22. RTOS function to run a periodic task.*

---

**Example 2.2.** Design a BSP for the LEDs.

**Solution:** Again, we need to separate what the system does from how it does it. We can turn LEDs on and off. In this example, the four LEDs constitute one 4-bit device, so we will organize the solution in that manner, as shown in Program 2.23. Again, we have hidden from the user the fact that we are running on a TM4C using Port D.

```

#define LEDS (*(volatile uint32_t *)0x4000703C)
//----- OS_LEDInit -----
// Initialize the set of 4 LEDs
// Input: none
// Output: none
void OS_LEDInit(void){ volatile uint32_t delay;
    SYSCTL_RCGCGPIO_R |= 0x08; // activate port D
    delay = SYSCTL_RCGCGPIO_R; // allow time for clock to stabilize

```

```

GPIO_PORTD_DIR_R |= 0x0F; // make PD3-0 out
GPIO_PORTD_AFSEL_R &= ~0x0F; // regular port function
GPIO_PORTD_DEN_R |= 0x0F; // enable digital I/O on PD3-0
}
//----- OS_LED_Out -----
// Output to the 4 LEDs
// Input: number from 0 to 15, specifying which LEDs are on and off
// Output: none
void OS_LEDOut(uint32_t number){
    LEDs = number; // friendly access
}

```

*Program 2.23. BSP for four LEDs.*

---

## 2.11.2 Path Expression

**Path expression** is a formal mechanism to specify the correct calling order in a group of related functions. Consider a UART device driver with 4 functions, the prototypes are

```

void UART_Init(void); // Initialize Serial port
char UART_InChar(void); // Wait for new serial port input
void UART_OutChar(char data); // Output 8-bit to serial port
void UART_Close(void); // Shut down serial port

```

It is obvious that you should not attempt to input/output until the UART is initialized. In this problem, we will go further and actually prevent the user from executing **UART\_InChar** and **UART\_OutChar** before executing **UART\_Init**. A directed graph is a general method to specify the valid calling sequences (Figure 2.32). An arrow represents a valid calling sequence within the path expression. The system “state” is determined by the function it called last. For this example, we begin in the closed state, because the UART is initially disabled. The tail of an arrow touches the function we called last, and the head of an arrow points to a function that we are allowed to call next. In this method, a calling sequence is valid if there is sequence of arrows to define it. For example, these calling sequences are valid

```

Init InChar InChar OutChar Close    d b e i j
Init OutChar OutChar OutChar OutChar d c g g g
Init Close Init InChar Close        d a d b h

```

On the other hand, the following calling sequences are illegal because each has no representative sequence of arrows

```

Init InChar Init OutChar Close    Can't initialize twice

```

Close            Can't close because already disabled  
 OutChar OutChar OutChar    Can't output without initialization

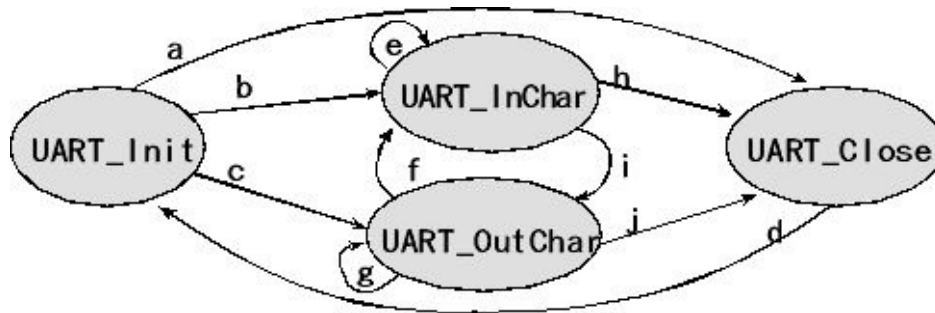


Figure 2.32. Directed graph showing path expression for the serial port driver.

A fast, but memory inefficient method, to represent a directed graph uses a square matrix. Since there are four functions, the matrix will be 4 by 4. The row number (0,1,2,3) will specify the current state (the function called last), and the column number (0,1,2,3) will specify the function that might be called next. The values in the matrix are true(1)/false(0) specifying whether or not the next function call is legal. Since there are 10 arrows in the directed graph, there will be exactly 10 true values in the matrix, one for each arrow. The remaining values will be false(0). Program 2.24 shows the data structure for the directed graph. At the beginning of each call to the serial port driver, the OS checks to verify the user has permission to execute that function. The global variable **State** defines the current state. For example, **Path[3][0]** will be true signifying it is OK to call **UART\_Init** if the UART is disabled. We assume there is an operating system function called **OS\_Kill()**, which should be called if a thread makes an illegal function call, destroying the thread because it has made a serious programming error.

```
int State=3; // start in the Closed state
int Path[4][4]={ /* Init InChar OutChar Close */
/*      column 0  1  2  3 */
/* Init  row 0*/ { 0, 1, 1, 1 },
/* InChar row 1*/ { 0, 1, 1, 1 },
/* OutChar row 2*/ { 0, 1, 1, 1 },
/* Close row 3*/ { 1, 0, 0, 0 }}
void UART_Init(void){
  if(Path[State][0]==0) OS_Kill(); // kill if illegal
  State = 0; // perform valid Init
  SYSCTL_RCGCUART_R |= 0x0001; // activate UART0
  SYSCTL_RCGCGPIO_R |= 0x0001; // activate port A
  UART0_CTL_R &= ~0x0001; // disable UART
  UART0_IBRD_R = 3; // int(6,000,000 / (16*115,200)) = int(3.2552)
  UART0_FBRD_R = 16; // int(0.2552 * 64 + 0.5) = 16
```

```

UART0_LCRH_R = 0x0070;    // 8-bit word length, enable FIFO
UART0_CTL_R = 0x0301;    // enable RXE, TXE and UART
GPIO_PORTA_AFSEL_R |= 0x03; // enable alt funct on PA1-0
GPIO_PORTA_DEN_R |= 0x03; // enable digital I/O on PA1-0
}
char UART_InChar(void){
    if(Path[State][1]==0) OS_Kill(); // kill if illegal
    State = 1;                // perform valid InChar
    while((UART0_FR_R&0x0010) != 0); // wait until RXFE is 0
    return((char)(UART0_DR_R&0xFF));
}
void UART_OutChar(char data){
    if(Path[State][2]==0) OS_Kill(); // kill if illegal
    State = 2;                // perform valid OutChar
    while((UART0_FR_R&0x0020) != 0); // wait until TXFF is 0
    UART0_DR_R = data;
}
void UART_Close(void){
    if(Path[State][3]==0) OS_Kill(); // kill if illegal
    State = 3;                // perform valid Close
    UART0_CTL_R &= ~0x0001;    // disable UART
}

```

*Program 2.24. Directed graph showing path expression for the serial port driver.*

---

## 2.12. Debugging

### 2.12.1. Functional Debugging

Functional debugging involves the verification of input/output parameters. It is a static process where inputs are supplied, the system is run, and the outputs are compared against the expected results. We will present seven methods of functional debugging.

1. *Single Stepping or Trace.* Many debuggers allow you to set the program counter to a specific address then execute one instruction at a time. **StepOver** will execute one instruction, unless that instruction is a subroutine call, in which case the simulator will execute the entire subroutine and stop at the instruction following the subroutine call. **StepOut** assumes the execution has already entered a function and will finish execution of the function and stop at the instruction following the function call.

2. *Breakpoints without filtering.* The first step of debugging is to **stabilize** the system with the bug. In the debugging context, we stabilize the problem by creating a test routine that fixes (or stabilizes) all the inputs. In this way, we can reproduce the exact inputs over and over again. Once stabilized, if we modify the program, we are sure that the change in our outputs is a function of the modification we made in our software and not due to a change in the input parameters. A **breakpoint** is a mechanism to tag places in our software, which when executed will cause the software to stop.

3. *Conditional breakpoints.* One of the problems with breakpoints is that sometimes we have to observe many breakpoints before the error occurs. One way to deal with this problem is the conditional breakpoint. Add a global variable called **count** and initialize it to zero in the ritual. Add the following conditional breakpoint to the appropriate location, and run the system again (you can change the 32 to match the situation that causes the error).

```
if(++count==32){  
    breakpoint(); // <= place breakpoint here  
}
```

Notice that the breakpoint occurs only on the 32<sup>nd</sup> time the break is encountered. Any appropriate condition can be substituted.

4. *Instrumentation: print statements.* The use of print statements is a popular and effective means for functional debugging. The difficulty with print statements in embedded systems is that a standard “printer” may not be available. Another problem with printing is that most embedded systems involve time-dependent interactions with its external environment. The print statement itself may so slow that the debugging instrument itself causes the system to fail. Therefore, the print statement is

usually intrusive. One exception to this rule is if the printing channel occurs in the background using interrupts, and the time between print statements ( $t_2$ ) is large compared to the time to execution one print ( $t_1$ ), then the print statements will be minimally intrusive. Nevertheless, this book will focus on debugging methods that do not rely on the availability of a printer.

5. *Instrumentation: dump into array without filtering.* One of the difficulties with print statements is that they can significantly slow down the execution speed in real-time systems. Many times the bandwidth of the print functions cannot keep pace with data being generated by the debugging process. For example, our system may wish to call a function 1000 times a second (or every 1 ms). If we add print statements to it that require 50 ms to perform, the presence of the print statements will significantly affect the system operation. In this situation, the print statements would be considered extremely intrusive. Another problem with print statements occurs when the system is using the same output hardware for its normal operation, as is required to perform the print function. In this situation, debugger output and normal system output are intertwined. To solve both these situations, we can add a debugger instrument that dumps strategic information into arrays at run time. Assume **P1** is an input and **P2** is an output port that are strategic to the system. The first step when instrumenting a dump is to define a buffer in RAM to save the debugging measurements. The **Debug\_Cnt** will be used to index into the buffers. **Debug\_Cnt** must be initialized to zero, before the debugging begins. The debugging instrument, shown in Program 2.25, saves the strategic data into the buffer. We can then observe the contents of the array at a later time. One of the advantages of dumping is that the JTAG debugging allows you to visualize memory while running.

```
#define SIZE 100
uint8_t Debug_Buffer[SIZE][2];
unsigned int Debug_Cnt=0;
void Debug_Dump(void){ // dump P1IN and P2OUT
    if(Debug_Cnt < SIZE){
        Debug_Buffer[Debug_Cnt][0] = P1IN;
        Debug_Buffer[Debug_Cnt][1] = P2OUT;
        Debug_Cnt++;
    }
}
```

*Program 2.25. Instrumentation dump without filtering.*

Next, you add **Debug\_Dump()** statements at strategic places within the system. You can either use the debugger to display the results or add software that prints the results after the program has run and stopped. In this way, you can collect information in the exact same manner you would if you were using print statements.

6. *Instrumentation: dump into array with filtering.* One problem with dumps is that they can generate a tremendous amount of information. If you suspect a certain

situation is causing the error, you can add a filter to the instrument. A filter is a software/hardware condition that must be true in order to place data into the array. In this situation, if we suspect the error occurs when the pointer nears the end of the buffer, we could add a filter that saves in the array only when data matches a certain condition. In the example shown in Program 2.26, the instrument saves the strategic variables into the buffer only when **P1.7** is high.

```
#define SIZE 100
uint8_t Debug_Buffer[SIZE][2];
unsigned int Debug_Cnt=0;
void Debug_FilteredDump(void){ // dump P1IN and P2OUT
    if((P1IN&0x80)&&(Debug_Cnt < SIZE)){
        Debug_Buffer[Debug_Cnt][0] = P1IN;
        Debug_Buffer[Debug_Cnt][1] = P2OUT;
        Debug_Cnt ++;
    }
}
```

*Program 2.26. Instrumentation dump with filter.*

7. *Monitor using the LED heartbeat.* Another tool that works well for real-time applications is the monitor. A **monitor** is an independent output process, somewhat similar to the print statement, but one that executes much faster and thus is much less intrusive. An LCD can be an effective monitor for small amounts of information if the time between outputs is much larger than the time to output. Another popular monitor is the LED. You can place one or more LEDs on individual otherwise unused output bits. Software toggles these LEDs to let you know what parts of the program are running. An LED is an example of a Boolean monitor or **heartbeat**. Assume an LED is attached to MSP432 Port 1 bit 0. Program 2.27 will toggle the LED.

```
#define LEDOUT (*((volatile uint8_t *) (0x42000000+32*0x4C02+4*0))
#define Debug_HeartBeat() (LEDOUT ^= 0x01)
```

*Program 2.27. An LED monitor, written as a C macro.*

Next, you add **Debug\_HeartBeat();** statements at strategic places within the system. Port 1 must be initialized so that bit 0 is an output before the debugging begins. You can either observe the LED directly or look at the LED control signals with a high-speed oscilloscope or logic analyzer. When using LED monitors, it is better to modify just the one bit, leaving the other 7 as is. In this way, you can have multiple monitors on one port.

**Checkpoint 2.14:** Write a debugging instrument that toggles Port 1 bit 3 (MSP432) or toggles Port A bit 3 (TM4C123).

**Observation:** For safety-critical systems we place debugging instruments into the system during testing. Once the system is certified functional, we deliver the system with the instruments still included. If we were to remove the debugging

instruments we would be obligated to retest the changed system.

## 2.12.2. Performance Debugging (FFT analysis)

Performance debugging involves the verification of timing behavior of our system. It is a dynamic process where the system is run, and the dynamic behavior of the system is compared against the expected results. We will present three methods of performance debugging, then apply the techniques to measure execution speed.

1. *Counting bus cycles.* For simple programs with little and no branching and for simple microcontrollers, we can estimate the execution speed by looking at the assembly code and adding up the time to execute each instruction.

2. *Instrumentation measuring with an independent counter.* SysTick is a 24-bit counter decremented every bus clock. It automatically rolls over when it gets to 0. If we are sure the execution speed of our function is less than  $2^{24}$  bus cycles, we can use this timer to collect timing information with only a minimal amount of intrusiveness.

3. *Instrumentation Output Port.* Another method to measure real-time execution involves an output port and an oscilloscope. Connect a microcontroller output bit to your scope. Add debugging instruments that set/clear these output bits at strategic places. Remember to set the port's direction register to 1. Assume an oscilloscope is attached to TM4C123 Port F bit 2. Program 2.28 can be used to set and clear the bit.

```
#define PF2 (*(volatile uint32_t *)0x40025010)
#define Debug_Set() (PF2 = 0x04)
#define Debug_Clear() (PF2 = 0x00)
```

*Program 2.28. Instrumentation output port, written as C macros.*

Next, you add **Debug\_Set();** and **Debug\_Clear();** statements before and after the code you wish to measure. Port F must be initialized so that bit 2 is an output before the debugging begins. You can observe the signal with a high-speed oscilloscope or logic analyzer.

```
    Debug_Set();
    Stuff(); // User code to be measured
    Debug_Clear();
```

To illustrate these methods, we will consider measuring the execution time of a 1024-element integer FFT function written by STMicroelectronics. For details on the FFT, see Section 6.5.

<b>grouploop</b>	<b>ADD</b>	<b>butternbr,butternbr,index,LSL#(16-</b>	<b>85</b>
<b>2)</b>			<b>1024</b>
<b>butterloop</b>	<b>BUTFLY4_V7</b>	<b>pssX,index,pssX,14,pssK</b>	<b>1024</b>
	<b>SUBS</b>	<b>butternbr,butternbr, #1&lt;&lt;16</b>	<b>1024</b>
	<b>BGE</b>	<b>butterloop</b>	<b>85</b>



<b>ADD</b>	<b>tmp, index, index, LSL#1</b>	<b>85</b>
<b>ADD</b>	<b>pssX, pssX, tmp</b>	<b>85</b>
<b>DEC</b>	<b>butternbr</b>	<b>85</b>
<b>MOVS</b>	<b>tmp2, butternbr, LSL#16</b>	<b>85</b>
<b>IT</b>	<b>NE</b>	<b>85</b>
<b>SUBNE</b>	<b>pssK, pssK, tmp</b>	<b>85</b>
<b>BNE</b>	<b>grouploop</b>	

*Program 2.29. A section of the FFT assembly listing and the number of times each instruction was executed.*

The first method is to count bus cycles using the assembly listing. This approach is only appropriate for very short programs. Counting cycles becomes difficult for long programs with many conditional branch instructions and macro expansions. The time to execute each assembly instruction can be found in the Cortex-M Technical Reference Manuals. Because of the complexity of the ARM Cortex-M processor, this method is only approximate. For example, the time to execute a divide depends on the data, and the time to execute a branch depends on the alignment of the instruction pipeline. A portion of the assembly output generated by the ARM Keil uVision compiler is presented on the left side of Program 2.29, and on the right is the number of times each instruction is executed. For most programs it is actually very difficult to get an accurate time measurement using this technique.

The second method uses an internal timer called SysTick. The 24-bit SysTick register ( **STCURRENT** ) that is automatically decremented at the bus frequency. When the counter hits zero, it is reloaded to 0xFFFFFFFF and continues to count down. If we are sure the function will complete in a time less than  $2^{24}$  bus cycles, then the internal timer can be used to measure execution speed empirically. The code in Program 2.30 first reads the SysTick counter, executes the function, and then reads the SysTick counter again. The elapsed time is the difference in the counter before and after. Since the execution speed may be dependent on the input data, it is often wise to measure the execution speed for a wide range of input parameters. There is a slight overhead in the measurement process itself. To be accurate, you could measure this overhead and subtract it off your measurements. In this case, a constant 6 is subtracted so that if the call to the function were completely removed the elapsed time would return 0. Notice that in this example, the total time including parameter passing is measured. Results show that this 1024-element FFT executes in 97,872 bus cycles.

```

uint32_t Before, Elapsed; // assume SysTick is initialized
int32_t x[1024], y[1024]; // assume x is filled with data
void FFT(void){
    Before = STCURRENT;
    cr4_fft_1024_stm32(y, x, 1024); // complex FFT of 1024 values
    Elapsed = (Before - STCURRENT - 6)&0x00FFFFFF;
}

```

## Program 2.30. Empirical measurement of dynamic efficiency (ProfileFFTxxx).

The third technique can be used in situations where a timer is unavailable or where the execution time might be larger than  $2^{24}$  counts. In this empirical technique we attach an unused output pin to an oscilloscope or to a logic analyzer. We will set the pin high before the call to the function and set the pin low after the function call. In this way a pulse is created on the digital output with duration equal to the execution time of the function. We assume Port F is available, and bit 2 is connected to the scope. By placing the function call in a loop, the scope can be triggered. With a storage scope or logic analyzer, the function need be called only once. Together with an oscilloscope or logic analyzer, Program 2.31 measures the execution time of the function `cr4_fft_1024_stm32` (Figure 2.33). We stabilize the system by calling it over and over. Using the scope, we can measure the width of the pulse on PF2, which will be execution time of the FFT. Running at 16 MHz, it takes about 6.08 ms to execute `cr4_fft_1024_stm32(y, x, 1024)`, which is about 97,300 bus cycles.

```
int main(void){ int32_t x[1024], y[1024];
  PortF_Init();    // Make PF2 output
  while(1){
    Debug_Set();   // set PF2 high
    cr4_fft_1024_stm32(y, x, 1024); // 1024 length FFT
    Debug_Clear(); // clear PF2 low
  }
}
```

## Program 2.31. Another empirical measurement of dynamic efficiency (ProfileFFTxxx).



Figure 2.33. Oscilloscope output measured from Program 2.31 using a PicoScope 2104, running at 16 MHz.

### 2.12.3. Debugging heartbeat

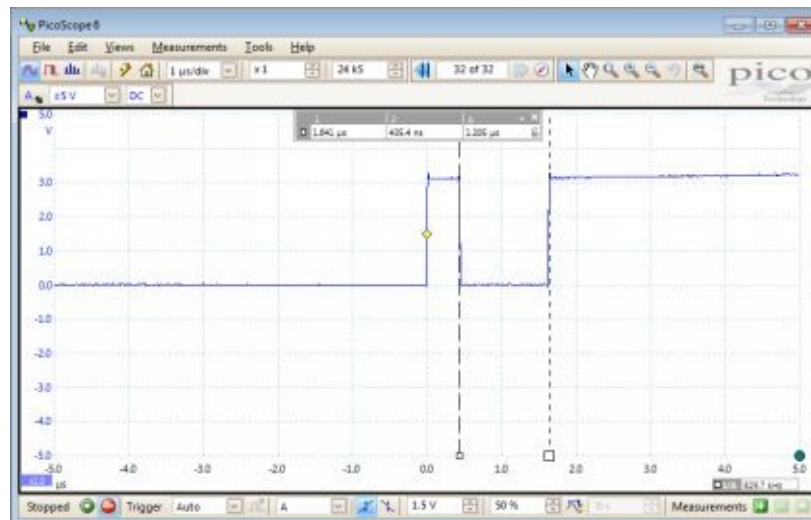
A debugging heartbeat would allow us to see if and when the ISR runs. If we toggle a pin once, we can measure when the interrupt occurred. If we toggle it three times,

like Program 2.5, we can also measure the execution time of the ISR. The first and second edges of PC5 signify the start of the ISR. The third edge occurs at the end of the ISR. The `PC5 ^= 0x20;` takes 4 instructions or 7 cycles

```
480D    LDR r0,[pc,#52] ; pointer to PC5
6BC0    LDR r1,[r0]    ; read PC5
F0800020 EOR r1,r1,#0x20 ; toggle
63C8    STR r1,[r0]    ; write PC5
```

These three debugging instruments add 21 bus cycles to each ISR. Thus, if the time between interrupts is large compared to these 21 cycles, this heartbeat will be minimally intrusive.

Figure 2.34 shows a zoomed in view of the profile pin measured during one execution of the SysTick ISR. The first two toggles signify the ISR has started. The time from second to third toggle illustrates the body of the ISR takes 1.2  $\mu$ s of execution time.



*Figure 2.34. Profile of a single execution of the SysTick ISR measured on a TM4C123 running at 16 MHz.*

Figure 2.35 shows a zoomed out view of the profile pin measured during multiple executions of the SysTick ISR. This measurement verifies the ISR runs every 100 ms. Because of the time scale, the three toggles appear as a single toggle. This **triple-toggle technique (TTT)** allows us to measure both the time to execution of one instance of the ISR and to measure the time between ISR executions.

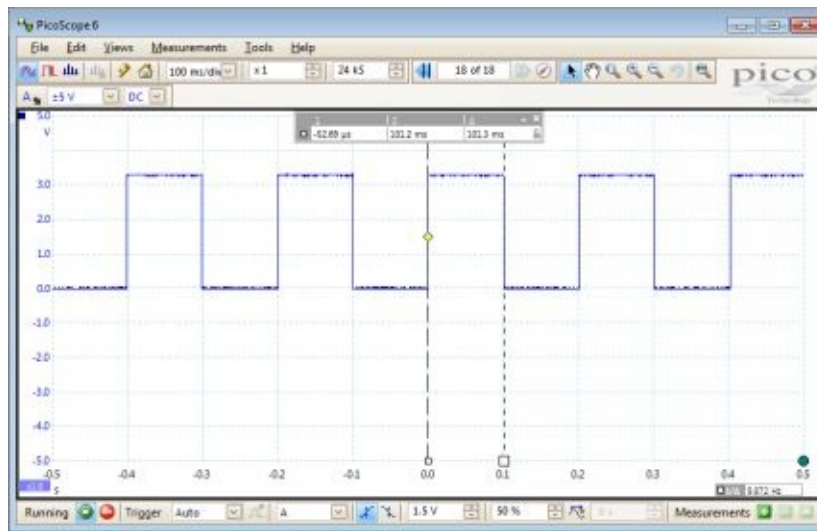


Figure 2.35. Profile of multiple executions of the SysTick ISR on a TM4C123 running at 16 MHz.

## 2.12.4. Profiling

**Profiling** is a type of performance debugging that collects the time history of program execution. Profiling measures where and when our software executes. It could also include what data is being processed. For example, if we could collect the time-dependent behavior of the program counter, then we could see the execution patterns of our software.

*Profiling using a software dump to study execution pattern.* In this section, we will discuss software instruments that study the execution pattern of our software. In order to collect information concerning execution we will add debugging instruments that save the time and location in arrays (Program 2.32). By observing these data, we can determine both a time profile (when) and an execution profile (where) of the software execution. Running this profile revealed the sequence of places as 0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, and 3. Each call to **Debug\_Profile** requires 32 cycles to execute. Therefore, this instrument is a lot less intrusive than a print statement.

```

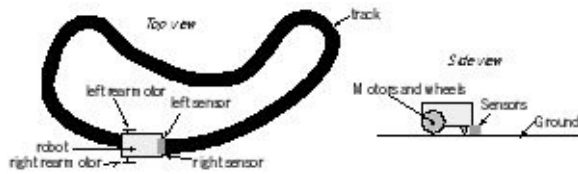
uint32_t Debug_time[20];
uint8_t Debug_place[20];
uint32_t n;
void Debug_Profile(uint8_t p){
    if(n < 20){
        Debug_time[n] = STCURRENT; // record current time
        Debug_place[n] = p;
        n++;
    }
}
uint32_t sqrt(uint32_t s){

```

```
uint32_t t;    // t*t becomes s
int n;        // loop counter
Debug_Profile(0);
t = s/10+1;   // initial guess
Debug_Profile(1);
for(n = 16; n; --n){ // will finish
    Debug_Profile(2);
    t = ((t*t+s)/t)/2;
}
Debug_Profile(3);
return t;
}
```

*Program 2.32. A time/position profile dumping into a data array.*

## 2.13. Exercises



**2.1** Draw a flowchart for a line-tracking robot. There are two inputs from the line sensors on the bottom, labeled **Right** and **Left**. If both sensors are true, then the robot is on the line. If **Right** is true and **Left** is false, the robot is veering off the left. If **Right** is false and **Left** is true, the robot is veering off the right. If both are false, the robot is off the line. There are two outputs to the motors labeled **GoRight** and **GoLeft**. If both outputs are true, the robot will go straight. If **GoRight** is true and **GoLeft** is false, the robot will turn left. If **GoRight** is false and **GoLeft** is true, the robot will turn right. If both outputs are false, then the robot will stop.

**2.2** A digital output of one microcontroller is connected to a digital input of another microcontroller. The output is configured with 2mA drive. The two microcontrollers share a common ground.

- When the output is high, which way does current flow along the wire between the pins?
- When the output is low, which way does current flow along the wire between the pins?
- When the output is high how much current flows? (less than  $2\mu\text{A}$ , exactly  $2\mu\text{A}$ , between exactly  $2\mu\text{A}$  and  $2\text{mA}$ , exactly  $2\text{mA}$ , or more than  $2\text{mA}$ ).
- When the output is low how much current flows? (less than  $2\mu\text{A}$ , exactly  $2\mu\text{A}$ , between exactly  $2\mu\text{A}$  and  $2\text{mA}$ , exactly  $2\text{mA}$ , or more than  $2\text{mA}$ ).

**2.3** Consider the situation in which the output of one digital circuit is connected to the inputs of two other digital circuits. There are no other connections on this signal, i.e., one output is tied to two inputs. The output specifications of the first circuit are  $V_{OH}$ ,  $V_{OL}$ ,  $I_{OH}$ , and  $I_{OL}$ . The input specifications of the second and third circuits are  $V_{IH}$ ,  $V_{IL}$ ,  $I_{IH}$ , and  $I_{IL}$ . These are the specifications, like you would find in a data sheet, not actual measurements of voltage and current like you would measure in lab with a DVM. Give the four **inequalities** relating these eight parameters ( $V_{OH}$ ,  $V_{OL}$ ,  $I_{OH}$ ,  $I_{OL}$ ,  $V_{IH}$ ,  $V_{IL}$ ,  $I_{IH}$ , and  $I_{IL}$ .) that must be true in order for the interface to operate properly. It may be necessary to also add numbers to these inequalities.

**2.4** Interface an LED to the microcontroller. Show the interface circuit, the initialization software, and two functions: one to turn it on and one to turn it off. Make the initialization friendly and use bit-specific addressing on the two functions.

- The LED parameters are  $I_d=1.5\text{mA}$  and  $V_d = 1.6\text{V}$

b) The LED parameters are  $I_d=2.5\text{mA}$  and  $V_d = 1.7\text{V}$

c) The LED parameters are  $I_d=25\text{mA}$  and  $V_d = 1.8\text{V}$

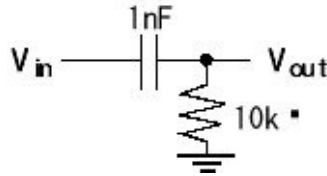
2.5 Write software that maintains hours (0 to 23), minutes (0 to 59) and seconds (0 to 59).

2.6 Rewrite the code in Program 2.5, so **Counts** is incremented every 1 second. Assume the bus clock is 50 MHz.

2.7 Rewrite the code in Program 2.5, so **SysTick\_Init** takes another input parameter, a call-by-reference to a function. This user defined function will be called in the ISR.

2.8 Write a formula relating baud rate (in bits/sec) to bandwidth (in bytes/sec) for a UART.

2.9 Sketch the step response of the following circuit. In particular draw the output wave as the input signal goes from 0 to 3.3 V.  $1\text{nF} \cdot 10\text{k}\Omega$  is 10  $\mu\text{sec}$ .



2.10 Consider the situation in which a software FIFO queue is used to buffer data between a main program and an output UART interrupt service routine (like Section 2.4). The main program calls **UART\_OutChar**, which in turn puts one byte into a software FIFO. The ISR is triggered when the UART hardware FIFO is not full. The UART ISR gets data from the software FIFO and puts it to the hardware FIFO. Experimental measurements show that the rate at which **UART\_OutChar** is called varies over time with an average rate of 1,000 times/sec. What does it mean? Choose A-F and briefly justify your selection.

A) The system could work, but the system is CPU bound

B) The system does not work, but could be corrected by increasing software FIFO size

C) The system could work, but the system is I/O bound

D) The system does not work, but could be corrected by increasing baud rate

E) The system works, but the software FIFO is not needed and could be replaced by a global variable

F) The system could work, but interrupts are not needed in this system

a) The UART baud rate is 5,000 bits/sec.

b) The UART baud rate is 100,000 bits/sec.

2.11 UART interrupts are armed so that interrupts occur when new data arrives into the microcontroller (like Section 2.4). Consider the situation in which a FIFO queue is used to buffer data between the receiverISR and the main program. The **UART0\_Handler** reads **UART0\_DR\_R** and saves the data by calling **RxFifo\_Put**. When the main program wants input it calls **UART\_InChar**, which in turn calls **RxFifo\_Get**. Experimental observations show the software FIFO

is usually empty, and has at most 3 elements. What does it mean? Choose A-F and briefly justify your selection.

- A) The system is CPU bound
- B) Bandwidth could be increased by increasing the software FIFO size
- C) The system is I/O bound
- D) The software FIFO could be replaced by a global variable
- E) The latency is small and bounded
- F) Interrupts are not needed in this system

**2.12** The main program synthesizes a waveform (defines a sequence of DAC output values) and a periodic output compare interrupt will output the data to the DAC separated by a fixed time. A software FIFO queue is used to buffer data between a main program (e.g., main program calls **DAC\_Out** , which in turn calls **Fifo\_Put** ). A timer interrupt service routine calls **Fifo\_Get** and actually writes to the DAC. At the beginning of the ISR, experimental observations show this software FIFO is usually empty, and has at most 3 elements. What does it mean? Choose A-F.

- A) The system not operating properly because it is CPU bound
- B) The system not operating properly but could be fixed by increasing software FIFO size
- C) The system is not operating properly because it is I/O bound
- D) The system is operating properly, but the software FIFO could be replaced by a global variable
- E) The system is operating properly, but bandwidth could be increased by increasing the timer interrupt rate
- F) The system is operating properly, but interrupts are not needed in this system

**2.13** Assume you are outputting a sin wave using an n-bit DAC. What is the maximum table size you could use, such that if you increased the size of the table beyond that size, there would be no more improvements in waveform quality?

**2.14** You wish to record sound. The frequency components you wish to analyze are 200 to 2000 Hz. The signal to noise ratio of your microphone is 50 dB. What ADC precision and sampling rate would you choose? Justify your answer.

**2.15** You wish to measure pressure from 0 to 300 mmHg with a resolution of 0.1 mmHg. The frequency components you wish to analyze are 0 to 200 Hz. What ADC precision and sampling rate would you choose? Justify your answer.

**2.16** You wish to measure distance (0 to 1 cm) using the 10-bit ADC on the microcontroller. The sampling rate is 1000 Hz. The frequencies of interest are 0 to 100 Hz. The ADC range is 0 to 3V. The sensitivity of the transducer and amplifier is 3V/cm. The signal to noise ratio of your analog circuit is 45 dB. Which of the following changes will improve the quality of the system the most? Justify your answer.

- A) increasing the ADC precision
- B) increasing the ADC sampling rate



- C) increasing the gain of the amplifier
- D) changing the transducer to one with less noise

**2.17** Most ADC codes are linear (Figure 2.26). Under what conditions would it be better to design a nonlinear ADC? Give an example application needing a nonlinear ADC.

**2.18** Define ADC sampling jitter. Estimate the sampling jitter of sampling in Program 2.20.

**2.19** Write a busy-wait function that samples ADC channels 1, 2, and 3. Show the initialization routine and the input function that returns all three samples. Design in such a way that it could operate concurrently with Program 2.20 sampling channel 0 in the background.

**2.20** Write an interrupting system that samples ADC channel 1 at 200 Hz. Show the initialization routine and the ISR. Data should be spooled into a software FIFO. Design in such a way that it could operate concurrently with Program 2.20 sampling channel 0 in the background. Channel 0 is not being sampled at 200 Hz.

**2.21** Write a busy-wait function that collects 1000 samples of ADC channel 0 at 500 kHz. Show the initialization routine and the input function that collects the 1000 samples. Assume there are no interrupts active and this is the only ADC task. Assume the bus clock is 50 MHz.

**2.22** Consider the following BSP function that outputs an 8-bit number to a port. Add debugging dumps that record the last 32 data values to the port.

<pre>// MSP432 version void BSP_Out(uint8_t data){   P2OUT = data; }</pre>	<pre>// TM4C version void BSP_Out(uint8_t data){   GPIO_PORTB_DATA_R =   data; }</pre>
--	--

Write the debugging instruments in such a way that data need not be shifted. For example, if **I** is the index at which the last value was written ( **I** ranges from 0 to 31), then **(I-n)&0x1F** will be the index of the  $n^{\text{th}}$  previous data.

# 3. Thread Management

## Chapter 3 objectives are to:

- Introduce real-time operating systems
- Discuss memory management and show solution to manage a heap
- Define threads and discuss multithreading
- Use spinlock semaphores to implement thread synchronization
- Present debugging techniques applicable for real-time systems

This chapter introduces real-time operating systems. The operating system must manage system resources and in this chapter we will begin with memory and the processor. We will develop a heap to provide dynamic memory allocation. Our first simple OS will employ a round robin preemptive scheduler.

---

## 3.1. Introduction to RTOS

### 3.1.1. Motivation

Consider a system with one input task, one output tasks and two non I/O tasks, as shown in Figure 3.1. The non-I/O tasks are called function3 and function4. Here are two possible ways of structuring a solution to the problem. The left side of the figure shows a busy-wait solution, where a single main program runs through the tasks by checking to see if the conditions for running the task have occurred. Busy-wait solution is appropriate for problems where the execution patterns for tasks are fixed and well-known, and the tasks are tightly coupled. An alternative to busy-wait is to assign one thread per task. Interrupt synchronization is appropriate for I/O even if the execution pattern for I/O is unknown or can dynamically change at run time. The difficulty with the single-foreground multiple-background threaded solutions developed without an operating system stems from answering, “How to handle complex systems with multiple foreground tasks that are loosely coupled?” A **real-time operating system** (RTOS) with a thread scheduler allows us to run multiple foreground threads, as shown on the right side of the figure. As a programmer we simply write multiple programs that all “look” like main programs. Once we have an operating system, we write Task1, Task2, Task3, and Task4 such that each behaves like a main program. One of the features implemented in an RTOS is a **thread scheduler**, which will run all threads in a manner that satisfies the constraints of the system.

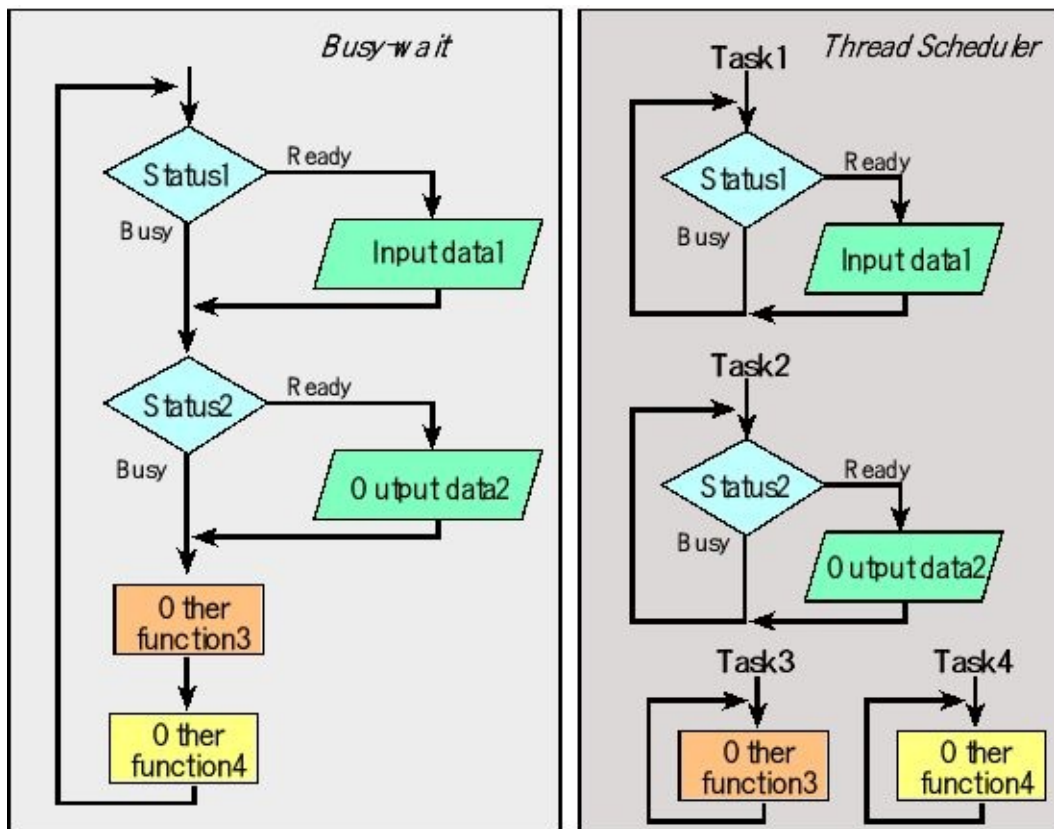


Figure 3.1. Flowcharts of a system with four loosely coupled tasks.

### 3.1.2. Parallel, distributed and concurrent programming

Many problems cannot be implemented using the single-threaded execution pattern. **Parallel programming** allows the computer to execute multiple threads at the same time. State-of-the-art multi-core processors can execute a separate program in each of its cores. **Fork** and **join** are the fundamental building blocks of parallel programming. After a fork, two or more software threads will be run in parallel. I.e., the threads will run simultaneously on separate processors. Two or more simultaneous software threads can be combined into one using a join, see Figure 3.2. Software execution after the join will wait until all threads above the join are complete.

As an analogy, if I want to dig a big hole in my back yard, I will invite three friends over and give everyone a shovel. The fork operation changes the situation from me working alone to four of us ready to dig. The four digging tasks are run in parallel. When the overall task is complete, the join operation causes the friends to go away, and I am working alone again. A complex system may employ multiple computers, each running its own software. We classify this configuration as **distributed programming**.

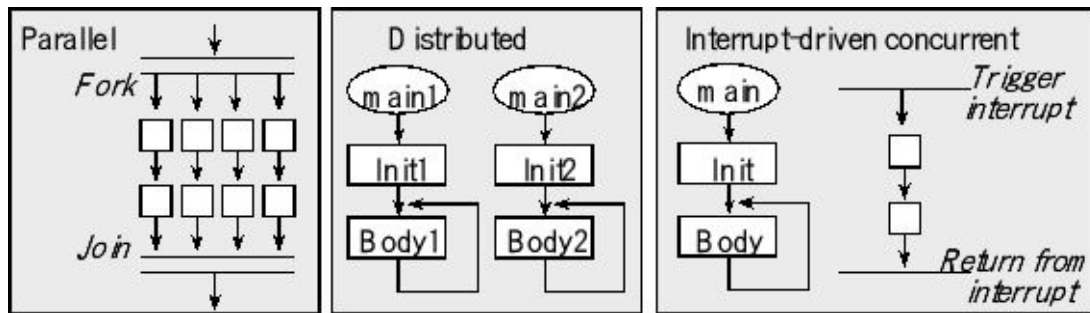


Figure 3.2. Flowchart symbols to describe parallel, distributed, and concurrent programming.

**Concurrent programming** allows the computer to execute multiple threads, but only one at a time. Interrupts are one mechanism to implement concurrency on real-time systems. **Interrupts** have a hardware trigger and a software action. An interrupt is a parameter-less subroutine call, triggered by a hardware event. The flowchart symbols for interrupts are also shown in Figure 3.2. The trigger is a hardware event signaling it is time to do something. Examples of interrupt triggers we will see in this book include new input data has arrived, output device is idle, and periodic event. The second component of an interrupt-driven system is the software action called an interrupt service routine (ISR). The foreground thread is defined as the execution of the main program, and the background threads are executions of the ISRs.

Consider the analogy of a farmer plowing a field. Plowing the field is like executing the main program in the foreground. You start plowing at one end of the field and travel back and forth across the land and basically plowing one parcel of land at a time in a sequential fashion. You might drive the tractor back to the barn, get some gas, then drive back to the field and continue plowing where you left off, which is analogous to a function call. Similarly, because of rocks or stumps you might have to plow a section over and over to get it right, which is analogous to a program loop. Even though you don't always drive in a straight line, you drive the tractor in a logical and well-defined sequence. How you drive the tractor while plowing the field is one process, defined by one algorithm. Conversely, if the chickens escape from their coop, you shut off the tractor, and race over to the coop. This is a real-time event, because you have a limited time to collect the chickens before they are lost or injured. When you are finished putting all the chickens back in the pen and fixing their fence, you get back on the tractor and continue plowing the field where you left off. The squawking of the chickens is analogous to hardware trigger and the chicken collection is like executing the ISR. Interrupts are hardware events that require software action. Understanding interrupts is critical for both designing a real-time operating system, as well as using one.

Continuing the farmer analogy, the farmer must perform many tasks, such as buying seed, plowing the field, planting the seed, harvesting the grain, and selling the grain. There may be many fields to manage, and each field may be in a different stage. If there is one farmer, he or she can only do one task at a time. He or she must develop a schedule so all tasks are completed in an effective manner. This scheduling is like

the one in a real-time operating system (RTOS). The RTOS is given many foreground tasks to perform and the rate to execute them. To be effective and efficient, just like the farmer, the RTOS needs to know how long each task requires to run, and what the relative priority is between tasks. The farm with many workers is analogous to an RTOS running on multiple processors. In this case, synchronization and communication are critical parts of the solution.

### 3.1.3. Introduction to threads

A **program** is a sequence of software commands connected together to affect a desired outcome. Programs perform input, make decisions, record information, and generate outputs. Programmers generate software using an editor with a keyboard and display. Programs are compiled and downloaded into the flash ROM of our microcontroller. Programs themselves are static and lifeless entities. However, when we apply power to the microcontroller, the processor executes the machine code of the programs in the ROM. A **thread** is defined as either execution itself or the action caused by the execution. Either way we see that threads are dynamic, and thus it is threads that breathe life into our systems. A thread therefore is a program in action, accordingly, in addition to the program (instructions) to execute it also has the state of the program. The thread state is captured by the current contents of the registers and the local variables, both of which are stored on the thread's stack.

For example, Figure 3.3 shows a system with four programs. We define Thread1 as the execution of Task1. Another name for thread is **light-weight process**. Multiple threads typically cooperate to implement the desired functionality of the system. We could use hardware-triggered interrupts to create multiple threads. However, in this class the RTOS will create the multiple threads that make up our system. Figure 3.3 shows the threads having separate programs. All threads do have a program to execute, but it is acceptable for multiple threads to run the same program. Since each thread has a separate stack, its local variables are private, which means it alone has access to its own local variables.

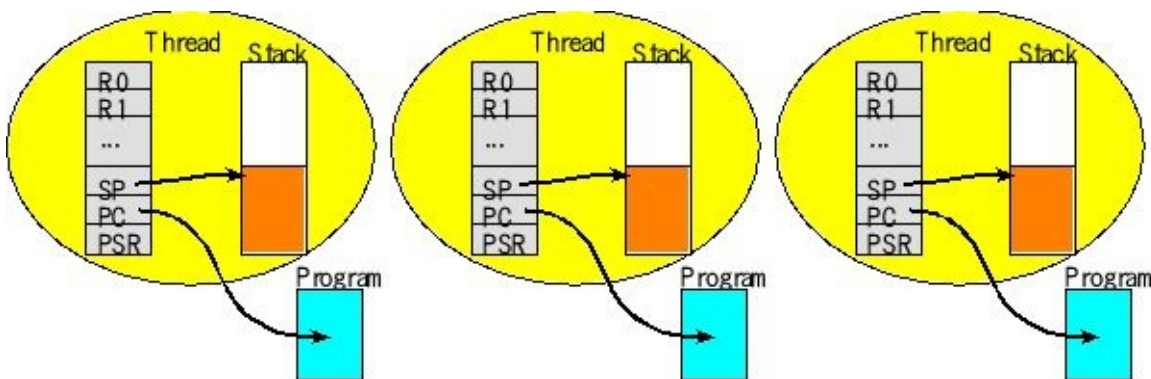


Figure 3.3. Each thread has its own registers and stack.

It looks like in Figure 3.3 that threads have physically separate registers. The stacks

will be physically separate, but in reality there is just one set of registers that is switched between the threads as the thread scheduler operates. The thread switcher will suspend one thread by pushing all the registers on its stack, saving the SP, changing the SP to point to the stack of the next thread to run, then pulling all the registers off the new stack.

Since threads interact for a common goal, they do share resources such as global memory, and I/O devices (Figure 3.4). However, to reduce complexity it is the best to limit the amount of sharing. It is better to use a well-controlled means to pass data and synchronize threads.

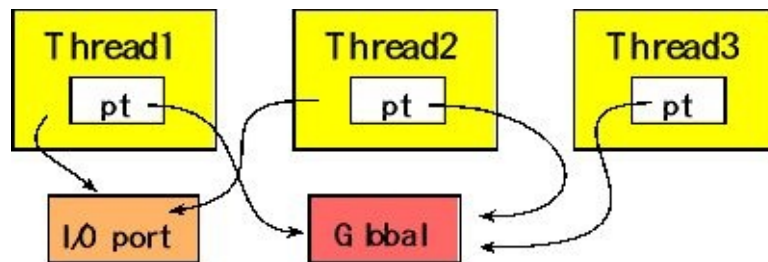


Figure 3.4. Threads share global memory and I/O ports.

Some simple examples of multiple threads are the interrupt-driven I/O. In each of these examples, the background thread (interrupt service routine) executes when the I/O device is done performing the required I/O operation. A single foreground thread (main program) executes during the times when no interrupts are needed. A global data structure is used to communicate between threads. Notice that data stored on the stack or in registers by one thread are not accessible by another thread.

**Checkpoint 3.1:** What is the difference between a program and a thread?

**Checkpoint 3.2:** Why can't threads pass parameters to each other on the stack like regular functions do? How do threads communicate with each other?

One way to classify threads is according to how often they are run. A **periodic thread** is one that runs at a fixed time interval. ADC sampling, DAC outputs, and digital control are examples of periodic tasks. The RTOS is responsible for scheduling periodic threads. An **aperiodic thread** is one that runs often, but the times when it needs run cannot be anticipated. Threads that are attached to human input will fall into this category. A **sporadic thread** is one that runs infrequently or maybe never at all, but is often of great importance. Examples of sporadic threads that have real-time requirements include power failure, CO warning, temperature overheating, and computer hardware faults.

A second way to classify threads is according to the activity that triggers the thread's execution. An **event thread** is triggered by an external event like the hardware timer, input device or output device. The external event creates the thread, the thread services that need, and then the thread is dismissed. A typical event thread is the execution of an interrupt service routine. A periodic thread can be classified as an event thread triggered by a timer. A **main thread** on the other hand is like a main

program, it runs for a long time performing tasks like input, storage, decisions, and output. Main threads can be thought of as cycle-stealing threads because they run when there are no events to service.

### 3.1.4. States of a main thread

A main thread can be in one of four states, as shown in Figure 3.5. The arrows in Figure 3.5 describe the condition causing the thread to change states. In this chapter, threads oscillate between the active and run states. To simplify the OS, we will create all main threads at initialization and these main threads will never block, sleep, or die.

A main thread is in the **run state** if it is currently executing. On a microcontroller with a single processor like the Cortex M, there can be at most one thread running at a time. As computational requirements for an embedded system rise, we can expect microcontrollers in the future to have multicore processors, like the ones seen now in our desktop PC. For a multicore processor, there can be multiple threads in the run state.

A main thread is in the **active state** if it is ready to run but waiting for its turn. A simple OS does not have sleeping or blocking; there will be one running thread and the other threads are active.

Sometimes a main thread needs to wait for a fixed amount of time. The OS will not run a main thread if it is in the **sleep state**. After the prescribed amount of time, the OS will make the thread active again. Sleeping would be used for tasks that are not real-time. Sleeping will be presented later in Section 4.4.

A main thread is in the **blocked state** when it is waiting for some external event like input/output (keyboard input available, printer ready, I/O device available.) We will implement blocking in the next chapter.

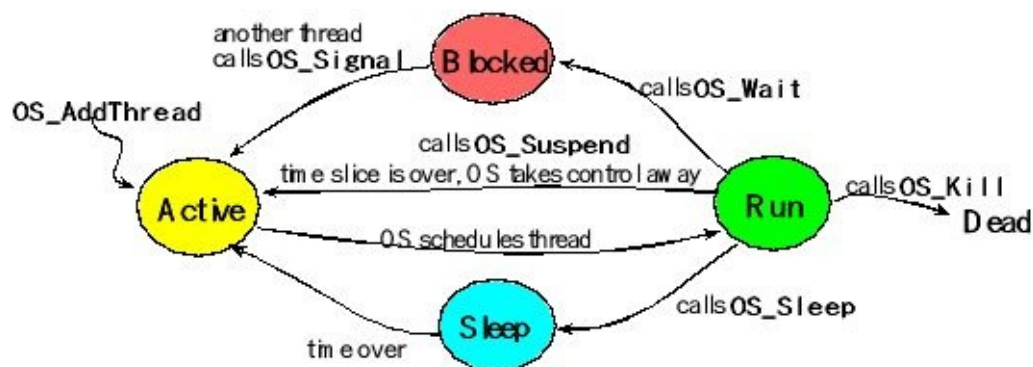


Figure 3.5. A main thread can be in one of four states.

The OS manages the execution of threads. An important situation to manage is when a thread is stuck and cannot make progress. For example, a thread may need data from another thread, a thread may be waiting on I/O, or a thread may need to wait for a specified amount of time. To be more efficient, when a thread is waiting because it



cannot make progress it will **block**, meaning it will not run until the time at which it can make progress. Similarly, to improve efficiency, when a thread needs to wait for a prescribed amount of time, it will **sleep**, meaning it will not run until the elapsed wait time has passed. Blocking and sleeping will free up the processor to perform actual work. A simple OS without blocking and sleeping must simply spin while the thread is waiting on an event. A thread that is spinning remains in the active state, and wastes its entire time slice checking the condition over and over.

### 3.1.5. Real-time systems

Designing a RTOS requires many decisions to be made. Therefore, it is important to have performance criteria with which to evaluate one alternative to another. A common performance criterion used in Real-Time Systems is **Deadline**, a timing constraint with many definitions in the literature. In this class we will define specific timing constraints that apply to design of embedded systems. **Bandwidth** is defined as the information rate. It specifies the amount of actual data per unit time that are input, processed, or output.

In a real-time system operations performed must meet logical correctness and also be completed on time (i.e., meet timing constraints). Non real-time systems require logical correctness but have no timing requirements. The tolerance of a real-time system towards failure to meet the timing requirements determines whether we classify it as **hard real time**, **firm real time**, or **soft real time**. If missing a timing constraint is unacceptable, we call it a hard real-time system. In a firm real-time system, the value of an operation completed past its timing constraint is considered zero but not harmful. In a soft real-time system, the value of an operation diminishes the further it completes after the timing constraint.

*Hard real time:* For example, if the pressure inside a module in a chemical plant rises above a threshold, failure to respond through an automated corrective operation of opening a pressure valve within a timing constraint can be catastrophic. The system managing the operations in such a scenario is a hard real-time operating system.

*Firm real time:* An example of a firm real-time system is a streaming multimedia communication system where failure to render one video frame on time in a 30 frames per second stream can be perceived as a loss of quality but does not affect the user experience significantly.

*Soft real time:* An example of a soft real-time system is an automated stock trading system where excessive delay in formulating an automated response to buy/sell may diminish the monetary value one can gain from the trade. The delivery of email is usually soft real time, because the value of the information reduces the longer it takes.

**Observation:** Please understand that the world has not reached consensus of the

definitions of hard, firm and soft. Rather than classify names to the real-time system, think of this issue as a continuum. There is a continuous progression of the consequence of missing a deadline: catastrophic (hard) → zero effect and no harm (firm) → still some good can come from finishing after deadline (soft). Similarly: there is a continuous progression for the value of missing a deadline: negative value (hard), zero value (firm) and some but diminishing positive value (soft).

To better understand real-time systems, **timing constraints** can be classified into two types. The first type is **event-response**. The event is a software or hardware trigger that signifies something important has occurred and must be handled. The response is the system's reaction to that event. Examples of event-response tasks include:

Operator pushes a button	->	Software performs action
Temperature is too hot	->	Turn on cooling fan
Supply voltage is too low	->	Activate back up battery
Input device has new data	->	Read and process input data
Output device is idle	->	Perform another output

The specific timing constraint for this type of system is called **latency**, which is the time between the event and the completion of the response. Let  $E_i$  be the times that events occur in our system, and  $T_i$  be the times these events are serviced. Latency is defined as

$$\Delta_i = T_i - E_i \text{ for } i = 0, 1, 2, \dots, n-1$$

where  $n$  is the number of measurements collected. The timing constraint is the maximum value for latency,  $\Delta_n$ , that is acceptable. In most cases, the system will not be able to anticipate the event, so latency for this type of system will always be positive.

A second type of timing constraint occurs with **prescheduled** tasks. For example, we could schedule a task to run periodically. If we define  $f_s$  as the desired frequency of a periodic task, then the desired period is  $\Delta t = 1/f_s$ . Examples of prescheduled tasks include:

Every 30 seconds	->	Software checks for smoke
At 22 kHz	->	Output new data to DACs creating sound
At 1 week, 1 month, 1 year	->	Perform system maintenance
At 300 Hz	->	Input new data from ADC measuring EKG
At 6 months of service	->	Deactivate system because it is at end of life

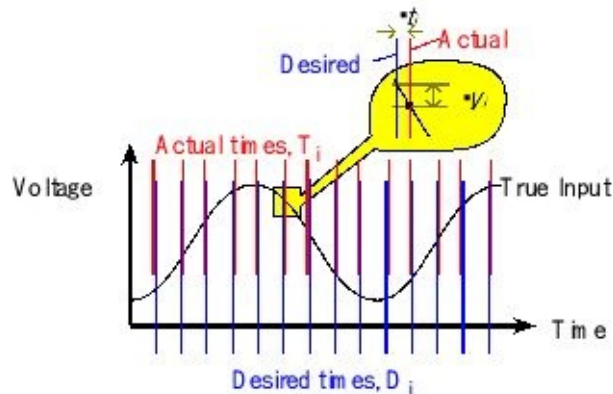
For periodic, the desired time to run the  $i$ 'th periodic instance of the task is given as

$$D_i = T_0 + i * \Delta t \text{ for } i = 0, 1, 2, \dots, n-1$$

where  $T_0$  is the starting time for the system. For prescheduled tasks, we define **jitter** as the difference between desired time a task is supposed to run and the actual time it is run. Let  $T_i$  be the actual times the task is run, so in this case jitter is

$$\delta t_i = T_i - D_i \quad \text{for } i = 0, 1, 2, \dots, n-1$$

Notice for prescheduled tasks the jitter can be positive (late) or negative (early), see Figure 3.6. For some situations running the task early is acceptable but being late is unacceptable. If I have the newspaper delivered to my door each morning, I do not care how early the paper comes, as long as it arrives before I wake up. In this case, the timing constraint is the maximum value for jitter  $\delta t_i$  that is acceptable.



*Figure 3.6. Effect of jitter on sampled data. True input is a sinusoidal. Blue lines depict when the voltage should be sampled. Red lines depict when the voltage was actually sampled. There is time jitter such that every other sample is early and every other sample is late. In the zoomed in portion this sample is late; the consequence of being late is the actual sampled data is lowered than the correct value. Sampling jitter causes noise in the data.*

On the other hand, for some situations, it is unacceptable to be early and it is acceptable to be late. For example, with tasks involving DACs and ADCs, as shown in Figure 3.6, we can correlate voltage error in the signal to time jitter. If  $dV/dt$  is the slew rate (slope) of the voltage signal, then the voltage error (noise) caused by jitter is

$$\delta V_i = \delta t_i * dV/dt \text{ for } i = 0, 1, 2, \dots, n-1$$

The error occurs because we typically store sampled data in a simple array and assume it was sampled at  $f_s = 1/\Delta t$ . I.e., we do not record exactly when the sample was actually performed.

For cases where the starting time,  $T_0$ , does not matter, we can simplify the analysis by looking at time differences between when the task is run,  $\Delta T_i = (T_i - T_{i-1})$ . In this case, jitter is simply

$$\delta t_i = \Delta T_i - \Delta t \quad \text{for } i = 0, 1, 2, \dots, n-1$$

We will classify a system with periodic tasks as real-time if the jitter is always less than a small but acceptable value. In other words, the software task always meets its timing constraint. More specifically, we must be able to place an upper bound,  $k$ , on the time jitter.

$$-k \leq \delta t_i \leq +k \quad \text{for all } i$$

For a hard real-time system, we are interested in the worst case. So we measure

$Min = \text{minimum } \delta t_i \text{ for all measurements } i$

$Max = \text{maximum } \delta t_i \text{ for all measurements } i$

$Jitter = Max - Min = (\text{maximum } \delta t_i - \text{minimum } \delta t_i)$

In most situations, the time jitter will be dominated by the time the microcontroller runs with interrupts disabled. For lower priority interrupts, it is also affected by the length and frequency of higher priority interrupt requests.

To further clarify this situation, we must clearly identify the times at which the  $T_i$  measurements are collected. We could define this time as when the task is started or when the task is completed. When sampling an ADC, the important time is when the ADC sampling is started. More specifically, it is the time the ADC sample/hold module is changed from sample to hold mode. This is because the ADC captures or latches the analog input at the moment the sample/hold is set to hold. For tasks with a DAC, the important time is when the DAC is updated. More specifically, it is the time the DAC is told to update its output voltage.

In this class, we use the term **real-time** and **hard real-time** to mean the same thing. Real-time for event-response tasks means the system has small and bounded latency. Real-time for periodic tasks means the system has small and bounded jitter. In other words, a real-time operating system (RTOS) is one that guarantees that the difference between when tasks are supposed to run and when they actually are run is short and bounded.

**Checkpoint 3.3:** Consider a task that inputs data from the serial port. When new data arrives the serial port triggers an event. When the software services that event, it reads and processes the new data. The serial port has hardware to store incoming data (2 on the MSP432, 16 on the TM4C123) such that if the buffer is full and more data arrives, the new data is lost. Is this system hard, firm, or soft real time?

**Checkpoint 3.4:** Consider a hearing aid that inputs sounds from a microphone, manipulates the sound data, and then outputs the data to a speaker. The system usually has small and bounded jitter, but occasionally other tasks in the hearing aid cause some data to be late, causing a noise pulse on the speaker. Is this system hard, firm or soft real time?

**Checkpoint 3.5:** Consider a task that outputs data to a printer. When the printer is idle the printer triggers an event. When the software services that event, it sends more data to the printer. Is this system hard, firm or soft real time?

### 3.1.6. Producer/Consumer problem using a mailbox

One of the classic problems our operating system must handle is communication between threads. We define a **producer** thread as one that creates or produces data. A **consumer** thread is a thread that consumes (and removes) data. The communication mechanism we will use in this chapter is a mailbox (Figure 3.7). The mailbox has a *Data* field and a *Status* field. Mailboxes will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be carefully shared by more than one task. The advantage of using a structure like a mailbox for a data flow problem is that we can decouple the producer and consumer threads. In the next chapter, we will replace the mailbox with a first in first one (FIFO) queue. The use of a FIFO can significantly improve system performance.

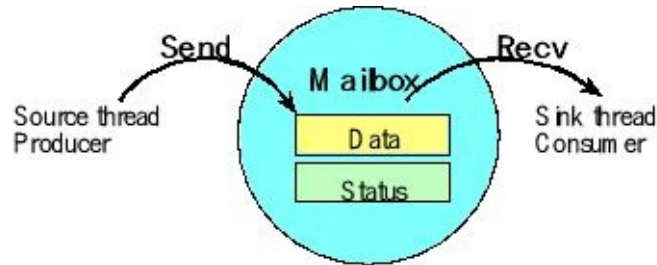


Figure 3.7. The mailbox is used to send data from the producer thread to the consumer thread.

There are many producer/consumer applications in the field of embedded systems. In Table 3.1 the threads on the left are producers that create data, while the threads on the right are consumers that process data.

Source/Producer	Sink/Consumer
Keyboard input	Program that interprets
Software that has data	Printer output
Software sends message	Software receives message
Microphone and ADC	Software that saves sound data
Software that has sound data	DAC and speaker

**Table 3.1. Producer consumer examples.**

Figure 3.8 shows how one could use a mailbox to pass data from a background thread (interrupt service routine) to a foreground thread (main program) if there were

no operating system.

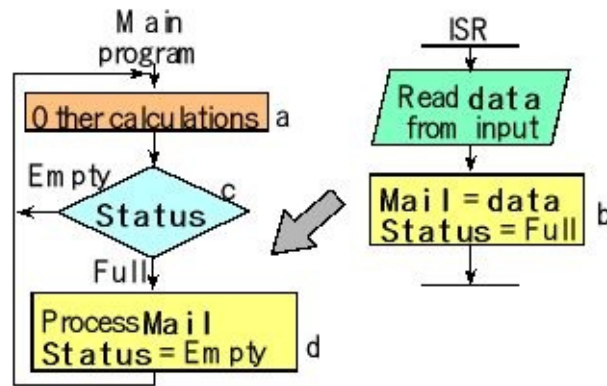


Figure 3.8. Use of a mailbox without an operating system.

**Checkpoint 3.6:** What happens if the ISR in Figure 3.8 runs twice before the main program has a chance to read and process the Mail?

### 3.1.7. Scheduler

A **scheduler** is an OS function that gives threads the notion of **Concurrent processing** where multiple threads are active. If we look from a distance (zoom out in time) it appears they are running simultaneously, when in fact only one thread is running at any time. On the Cortex-M with one processor only a single thread can run at any given time while other ready threads contend for processing. The scheduler therefore runs the ready threads one by one, switching between them to give us the illusion that all are running simultaneously.

In this class, the OS will schedule both main threads and event threads. However, in this section we will discuss scheduling main threads. To envision a scheduler, we first list the main threads that are ready to run. When the processor is free, the scheduler will choose one main thread from the ready list and cause it to run. In a **preemptive scheduler**, main threads are suspended by a periodic interrupt, the scheduler chooses a new main thread to run, and the return from interrupt will launch this new thread. In this situation, the OS itself decides when a running thread will be suspended, returning it to the active state. In Program 3.1, there exist four threads as illustrated in Figure 3.9. The preemptive scheduler in the RTOS runs the four main threads concurrently. In reality, the threads are run one at time in sequence.

<pre> void Task1(void){   Init1();   while(1){     if(Status1())       Input1();   } }           </pre>	<pre> void Task2(void){   Init2();   while(1){     if(Status2())       Output2();   } }           </pre>	<pre> void Task3(void){   Init3();   while(1){     function3();   } }           </pre>	<pre> void Task4(void){   Init4();   while(1){     function4();   } }           </pre>
---	--	--	--

Program 3.1. Four main threads run concurrently using a preemptive scheduler.

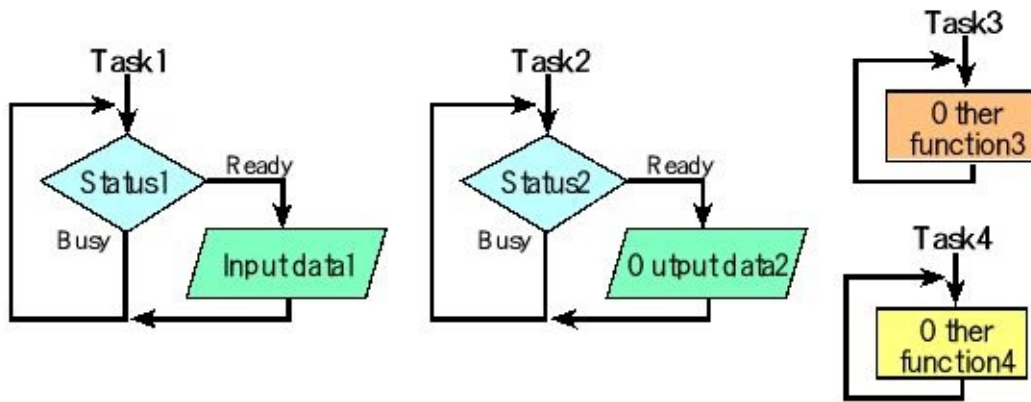


Figure 3.9. Four main threads.

In a **cooperative** or **nonpreemptive scheduler**, the main threads themselves decide when to stop running. This is typically implemented by having a thread call a function like **OS\_Suspend**. This function will suspend the running thread (putting the old thread in the **Active** state), run the scheduler (which chooses a new thread), and launch the new thread. The new thread is now in the **Run** state. Although easy to implement because it doesn't require interrupts, a cooperative scheduler is not appropriate for real-time systems. In Program 3.2, the cooperative scheduler runs the four main threads in a cyclic manner.

<pre>void Task1(void){   Init1();   while(1){     if(Status1()){       Input1();     }     OS_Suspend();   } }</pre>	<pre>void Task2(void){   Init2();   while(1){     if(Status2()){       Output2();     }     OS_Suspend();   } }</pre>	<pre>void Task3(void){   Init3();   while(1){     function3();     OS_Suspend();   } }</pre>	<pre>void Task4(void){   Init4();   while(1){     function4();     OS_Suspend();   } }</pre>
--	---	--	--

Program 3.2. Four threads run in a cooperative manner.

There are many scheduling algorithms one can use to choose the next thread to run. A **round robin scheduler** simply runs the ready threads in circular fashion, giving each the same amount of time to execute. A **weighted round robin scheduler** runs the ready threads in circular fashion, but gives threads unequal weighting. One way to implement weighting is to vary the time each thread is allowed to run according to its importance. Another way to implement weighting is to run important threads more often. E.g., assume there are three threads 1 2 3, and thread 1 is more important. We could run the threads in this repeating pattern: 1, 2, 1, 3, 1, 2, 1, 3... Notice that very often time slice is given to thread 1. In this simple example, Thread 1 receives 50% of the processor time, and threads 2 and 3 each receive 25%. A **priority scheduler** assigns each thread a priority number (e.g., 1 is the highest). Two or more threads can have the same priority. A priority-2 thread is run only if no priority-1 threads are ready to run. Similarly, we run a priority-3 thread only if no priority-1 or priority-2 threads are ready. If all threads have the same priority, then the scheduler reverts to a round-robin system. The advantage of priority is that we can reduce the latency

(response time) for important tasks by giving those tasks a high priority. The disadvantage is that on a busy system, low priority threads may never be run. This situation is called **starvation**.

Schedulers for real-time systems may use other metrics to decide thread importance/priority. A **deadline** is when a task should complete relative to when it is ready to run. The **time to deadline** is the time between now and the deadline. If you have a paper due on Friday, and it is Tuesday, the time-to-deadline is 3 days. Furthermore, we define **slack time** as the time-to-deadline minus the how long it will take to complete the task. If you have a paper due on Friday, it is Tuesday and it will take you one day to write the paper, your slack time is 2 days. Once the slack time becomes negative, you will miss your deadline. There are many other ways to assign priority:

- Minimize latency for real-time tasks
- Assign a dollar cost for delayed service and minimize cost
- Give priority to I/O bound tasks over CPU bound tasks
- Give priority to tasks that need to run more frequently
- Smallest time-to-deadline first
- Least slack time first

A thread's priority may be statically assigned or can be changed dynamically as the system progresses. An **exponential queue** is a dynamic scheduling algorithm, with varying priorities and time slices. If a thread blocks on I/O, its priority is increased and its time slice is halved. If it runs to completion of a time slice, its priority is decreased and its time slice is doubled.

Another dynamic scheduling algorithm uses the notion of **aging** to solve starvation. In this scheme, threads have a permanent fixed priority and a temporary working priority. The permanent priority is assigned according the rules of the previous paragraph, but the temporary priority is used to actually schedule threads. Periodically the OS increases the temporary priority of threads that have not been run in a long time. Once a thread is run, its temporary priority is reset back to its permanent priority.

Assigning priority to tasks according to how often they are required to run (their periodicity) is called a **Rate Monotonic Scheduler**. Assume we have  $m$  tasks that are periodic, running with periods  $T_j$  ( $0 \leq j \leq m-1$ ). We assign priorities according to these periods with more frequent tasks having higher priorities. Furthermore, let  $E_j$  be the maximum time to execute each task. Assuming there is little interaction between tasks, the **Rate Monotonic Theorem** can be used to predict if a scheduling solution exists. Tasks can be scheduled if

$$\sum_{j=0}^{m-1} \frac{E_j}{T_j} \leq m(2^{1/m} - 1)$$



and

$$\lim_{m \rightarrow \infty} m(2^{1/m} - 1) = \ln(2)$$

What this means is, as long as the total utilization of the set of tasks is below 69.32% ( $\ln(2) \approx 0.6932$ ) RMS will guarantee to meet all timing constraints. The practical application of the Rate Monotonic Theorem is extremely limited because most systems exhibit a high degree of coupling between tasks. Nevertheless, it does motivate a consideration that applies to all real-time operating systems. Let  $E_j$  be the time to execute each task, and let  $T_j$  be the time between executions of each task. In general,  $E_j/T_j$  will be the percentage of time Task  $j$  needs to run. The sum of these percentages across all tasks yields a parameter that estimates processor **utilization**.

$$\text{Average Utilization} \equiv \frac{\sum_{j=0}^{m-1} \text{ave } E_j}{\sum_{j=0}^{m-1} \text{ave } T_j}$$

$$\text{Maximum Utilization} \equiv \sum_{j=0}^{m-1} \max \frac{E_j}{T_j}$$

If utilization is over 100% there will be no solution. If utilization is below 5%, the processor may be too fast for your problem. The solution could be to slow down the clock and save power. As the sum goes over 50% and begins to approach 100%, it will be more and more difficult to schedule all tasks. The solution will be to use a faster processor or simplify the tasks. An effective system will operate in the 5 to 50% range.

**Checkpoint 3.7:** What happens if the average utilization is over 1?

**Checkpoint 3.8:** What happens if the average utilization is less than 1, but the maximum utilization is over 1?

---

## 3.2. Function pointers

As we work our way towards constructing an OS there are some advanced programming concepts we require the reader to be familiar with. One such concept is “function pointers”. Normally, when software in module A wishes to invoke software in module B, module A simply calls a function in module B. The function in module B performs some action and returns to A. At this point, typically, this exchange is complete. A **callback** is a mechanism through which the software in module B can call back a preset function in module A at a later time. Another name for callback is **hook**. To illustrate this concept, let module A be the user code and module B be the operating system. To setup a callback, we first write a user function (e.g., **CallMe** ), and then the user calls the OS passing this function as a parameter.

```
int count;
void CallMe(void){
    count++;
}
```

The OS immediately returns to the user, but at some agreed upon condition, the OS can invoke a call back to the user by executing this function.

As we initialize the operating system, the user code must tell the OS a list of tasks that should be run. More specifically, the user code will pass into the operating system pointers to user functions. In C on the Cortex M, all pointers are 32-bit addresses regardless of the type of pointer. A **function pointer** is simply a pointer to a function. In this book, all tasks or threads will be defined as void-void functions, like **CallMe**. In other words, threads take no inputs and return no output.

There are three operations we can perform on function pointers. The first is declaring a function pointer variable. Just like other pointers, we specify the type and add \* in front of the name. We think it is good style to include **p** , **pt** , or **ptr** in pointer names. The syntax looks like this

```
void (*TaskPt)(void);
```

Although the above line looks a little bit like a prototype, it is not a prototype. Rather this line creates a variable of type function pointer. We can read this declaration as **TaskPt** is a pointer to a function that takes no input and returns no output.

Just like other variables, we need to set its value before using it. To set a function pointer we assign it a value of the proper type. In this case, **TaskPt** is a pointer to a void-void function, so we assign it the address of a void-void function by executing this code at run time.

```
TaskPt = &CallMe; // TaskPt points to CallMe
```

Just like other pointers (to variables), to access what a pointer is pointing to, we dereference it using \*. In this case, to run the function we execute

```
*TaskPt()); // call the function to which it points
```

As an example, let's look at one of the features in the BSP package. The function **BSP\_PeriodicTask\_Init** will initialize a timer so a user function will run periodically. Notice the user function is called from inside the interrupt service routine.

```
void (*PeriodicTask)(void); // user function  
void BSP_PeriodicTask_Init(void(*task)(void), // user function  
    uint32_t freq, // frequency in Hz  
    uint8_t priority){ // priority  
// ...  
    PeriodicTask = task; // user function  
// ...  
}  
void T32_INT1_IRQHandler(void){  
    TIMER32_INTCLR1 = 0x00000001; // acknowledge Timer 1 interrupt  
    (*PeriodicTask()); // execute user task  
}
```

The user code creates a void-void function and calls **BSP\_PeriodicTask\_Init** to attach this function to the periodic interrupt:

```
BSP_PeriodicTask_Init(&checkboxbuttons, 10, 2);
```

Another application of function pointers is a hook. A **hook** is an OS feature that allows the user to attach functions to strategic places in the OS. Examples of places we might want to place hooks include: whenever the OS has finished initialization, the OS is running the scheduler, or whenever a new thread is created. To use a hook, the user writes a function, calls the OS and passes a function pointer. When that event occurs, the OS calls the user function. Hooks are extremely useful for debugging.

The compiler resolves addresses used in function calls during linking. Once you download the code, you cannot change it unless you reedit source code, recompile and redownload. Callbacks are a mechanism to change which function gets called dynamically, at run time. In a more complex system, the OS and the user code might not be compiled at the same time. One could compile and load the OS onto the system. Later, one compiles and loads the user code onto the same system. The two modules are then linked together using function pointers. For an example of this type of linking, see **OS\_AddThreads** later in the chapter.

---

## 3.3. Thread Management

### 3.3.1. Two types of threads

A fundamental concept in operating systems is the notion of an execution context referred to as a **thread**. We introduced threads and their components in Section 3.1.3, we will now look at the types of threads and how they are treated differently in the OS. We define two types of threads in this book. **Event threads** are attached to hardware and should execute on changes in hardware status. Examples include periodic threads that should be executed at a fixed rate (for example, data acquisition and control), input threads that should be executed when new data are available at the input device (like the operator pushed a button), and output threads that should be executed when the output device is idle and new data are available for output. They are typically defined as **void-void** functions. The time to execute an event thread should be short and bounded. In other words, event threads must execute and return. The time to execute an event thread must always be less than a small value (e.g., 10 $\mu$ s). In an embedded system without an OS, event threads are simply the interrupt service routines (ISRs). However, with a RTOS, we will have the OS manage the processor and I/O, and therefore the OS will manage the ISRs. The user will write the software executed as an event thread, but the OS will manage the ISR and call the appropriate event thread. Communication between threads will be managed by the OS. For example, threads could use a FIFO to pass data.

```
void periodicThread(void){ // called periodically  
    PerformTask();  
}  
void inputThread(void){ // new input is available  
    data = ReadInput(); // input data from hardware  
    Send(data); // pass data to other software  
}  
void outputThread(void){ // output is idle  
    data = Recv(); // get data from other software  
    WriteOutput(data); // output data to hardware  
}
```

The second type of thread is a **main thread**. Without an OS, embedded systems typically have one main program that is executed on start up. This main initializes the system and defines the high level behavior of the system. In an OS however, we will have multiple main threads. Main threads execute like main programs that never return. These threads execute an initialization once and then repeatedly execute a sequence of steps within a while loop. Here in this chapter, we will specify all the main threads at initialization and these threads will exist indefinitely. However, in later chapters we will allow main threads to be created during execution, and we

will allow main threads to be destroyed dynamically.

```

void mainThread(void){
  Init();
  while(1){
    Body();
  }
}

```

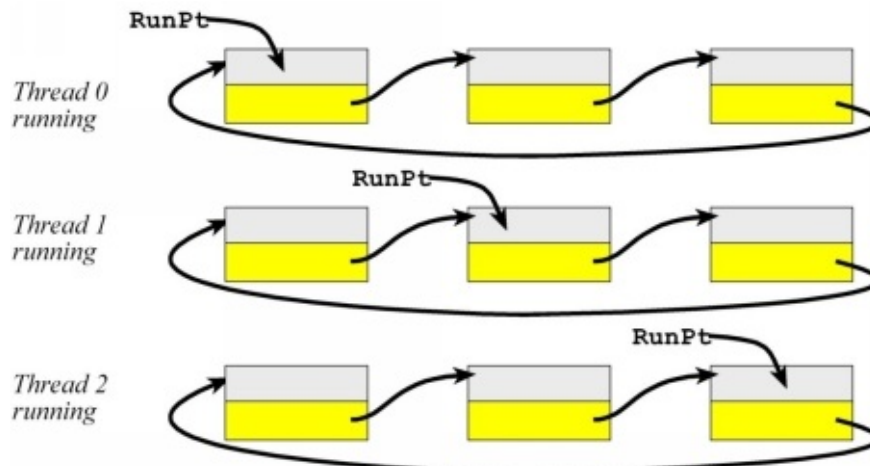
Table 3.2 compares event and main threads. For now, main threads will run indefinitely, but later in the class we will allow main threads to be terminated if their task is complete. It will be simpler if we will create all the main threads statically at the time the OS launches. To be more dynamic we will allow the user to create main threads dynamically at run time.

<i>Event Thread</i>	<i>Main Thread</i>
Triggered by hardware Must return	Created when OS launches Runs indefinitely
Short execution time	Unbounded execution time
No waiting	Allowed to wait
Finite number of loops (definite)	Indefinite or infinite loops

**Table 3.2. Comparison of event and main threads.**

### 3.3.2. Thread Control Block (TCB)

Figure 3.10 shows three threads. Each thread has a thread control block (TCB) encapsulating the state of the thread. For now, a thread's TCB we will only maintain a link to its stack and a link to the TCB of the next thread.



*Figure 3.10. Three threads have their TCBs in a circular linked list.*

The **RunPt** points to the TCB of the thread that is currently running. The **next** field is a pointer chaining all three TCBs into a circular linked list. Each TCB has an **sp** field. If the thread is running it is using the real SP for its stack pointer. However, the other threads have their stack pointers saved in this field. Other fields that define a thread's state such as, status, Id, sleeping, age, and priority will be added later. However, for your first RTOS, the **sp** and **next** fields will be sufficient. The scheduler traverses the linked list of TCBs to find the next thread to run.

In Figure 3.11 we illustrate how a round robin thread scheduler works. In this example there are three threads in a circular linked list. Each thread runs for a fixed amount of time, and a periodic interrupt suspends the running thread and switches **RunPt** to the next thread in the circular list. The scheduler then launches the next thread.

The **Thread Control Block** (TCB) will store the information private to each thread. There will be a TCB structure and a stack for each thread. While a thread is running, it uses the actual Cortex M hardware registers (Figure 3.11). Program 3.3 shows a TCB structure with the necessary components for three threads:

1. A pointer so it can be chained into a linked list
2. The value of its stack pointer

In addition to these necessary components, the TCB might also contain:

3. Status, showing resources that this thread has or wants
4. A sleep counter used to implement sleep mode
5. Thread number, type, or name
6. Age, or how long this thread has been active
7. Priority (not used in a round robin scheduler)

```
#define NUMTHREADS 3    // maximum number of threads
#define STACKSIZE 100  // number of 32-bit words in stack
struct tcb{
    int32_t *sp;    // pointer to stack, valid for threads not running
    struct tcb *next; // linked-list pointer
};
typedef struct tcb tcbType;
tcbType tcbs[NUMTHREADS];
tcbType *RunPt;
int32_t Stacks[NUMTHREADS][STACKSIZE];
```

*Program 3.3. TCBs for up to 3 threads, each stack is 400 bytes.*

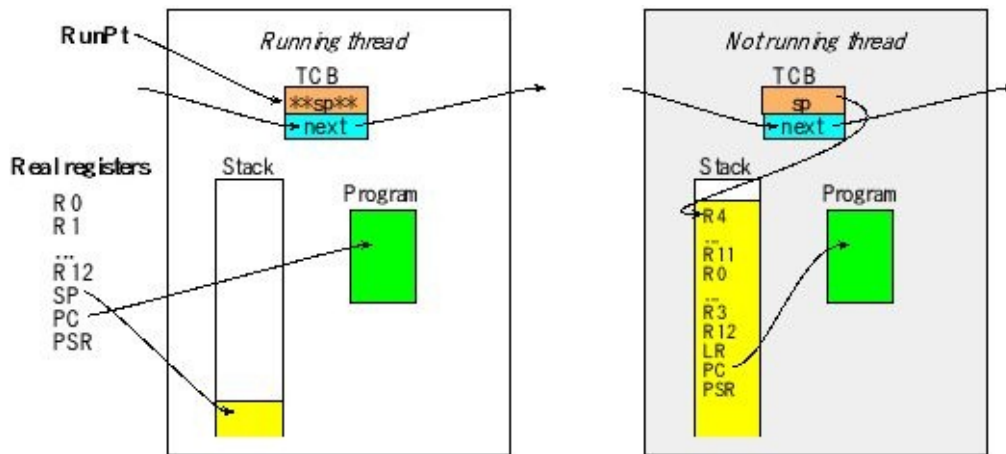


Figure 3.11. The running thread uses the actual registers, while the other threads have their register values saved on the stack. For the running thread the *sp* field is not valid, while the *sp* field on other threads points to the top of its stack.

### 3.3.3. Creation of threads

Program 3.4 shows how to create three TCBs that will run three programs. First, the three TCBs are linked in a circular list. Next the initial stack for each thread is created in such a way that it looks like it has been running already and has been previously suspended. The PSR must have the T-bit equal to 1 because the Arm Cortex M processor always runs in Thumb mode. The PC field on the stack contains the starting address of each thread. The initial values for the other registers do not matter, so they have been initialized to values that will assist in debugging. This idea came from the `os_cpu.c` file in Micrium  $\mu$ C/OS-II. The allocation of the stack areas must be done such that the addresses are double-word aligned.

```

void SetInitialStack(int i){
    tcbs[i].sp = &Stacks[i][STACKSIZE-16]; // thread stack pointer
    Stacks[i][STACKSIZE-1] = 0x01000000; // Thumb bit
    Stacks[i][STACKSIZE-3] = 0x14141414; // R14
    Stacks[i][STACKSIZE-4] = 0x12121212; // R12
    Stacks[i][STACKSIZE-5] = 0x03030303; // R3
    Stacks[i][STACKSIZE-6] = 0x02020202; // R2
    Stacks[i][STACKSIZE-7] = 0x01010101; // R1
    Stacks[i][STACKSIZE-8] = 0x00000000; // R0
    Stacks[i][STACKSIZE-9] = 0x11111111; // R11
    Stacks[i][STACKSIZE-10] = 0x10101010; // R10
    Stacks[i][STACKSIZE-11] = 0x09090909; // R9
    Stacks[i][STACKSIZE-12] = 0x08080808; // R8
    Stacks[i][STACKSIZE-13] = 0x07070707; // R7
    Stacks[i][STACKSIZE-14] = 0x06060606; // R6
    Stacks[i][STACKSIZE-15] = 0x05050505; // R5

```

```

Stacks[i][STACKSIZE-16] = 0x04040404; // R4
}
int OS_AddThreads(void(*task0)(void), void(*task1)(void),
                 void(*task2)(void)){
int32_t status;
status = StartCritical();
tcbs[0].next = &tcbs[1]; // 0 points to 1
tcbs[1].next = &tcbs[2]; // 1 points to 2
tcbs[2].next = &tcbs[0]; // 2 points to 0
SetInitialStack(0); Stacks[0][STACKSIZE-2] = (int32_t)(task0); // PC
SetInitialStack(1); Stacks[1][STACKSIZE-2] = (int32_t)(task1); // PC
SetInitialStack(2); Stacks[2][STACKSIZE-2] = (int32_t)(task2); // PC
RunPt = &tcbs[0]; // thread 0 will run first
EndCritical(status);
return 1; // successful
}

```

*Program 3.4. OS code used to create three active threads.*

Even though the thread has not yet been run, it is created with an initial stack that “looks like” it had been previously suspended by a SysTick interrupt. Notice that the initial value loaded into the PSR when the thread runs for the first time has T=1. Program 3.5 shows simple user software that can be run on this RTOS. Each thread increments a counter and toggles an output pin. The three counters should be approximately equal. Profile bit 0 toggles quickly while thread 0 is running. Profile bits 1 and 2 toggle when running threads 1 and 2 respectively.

```

void Task0(void){
    Count0 = 0;
    while(1){
        Count0++;
        Profile_Toggle0(); // toggle bit
    }
}
void Task1(void){
    Count1 = 0;
    while(1){
        Count1++;
        Profile_Toggle1(); // toggle bit
    }
}
void Task2(void){
    Count2 = 0;
    while(1){
        Count2++;

```



```

    Profile_Toggle2(); // toggle bit
}
}
#define THREADFREQ 500 // frequency in Hz
int main(void){
    OS_Init(); // initialize, disable interrupts
    Profile_Init(); // enable digital I/O on profile pins
    OS_AddThreads(&Task0, &Task1, &Task2);
    OS_Launch(BSP_Clock_GetFreq()/THREADFREQ); // interrupts enabled
    return 0; // this never executes
}

```

*Program 3.5. Example user code with three threads.*

### 3.3.4. Launching the OS

SysTick will be used to perform the preemptive thread switching. We will set the SysTick to the lowest level so we know it will only suspend foreground threads (Program 3.6).

```

void OS_Init(void){
    DisableInterrupts();
    BSP_Clock_InitFastest();// set processor clock to desired speed
}

```

*Program 3.6. RTOS initialization.*

To start the RTOS, we write code that arms the SysTick interrupts and unloads the stack as if it were returning from an interrupt (Program 3.7). The units of **theTimeSlice** are in bus cycles. The bus cycle time on the TM4C123 is 12.5ns, and on the MSP432 the bus cycle time is 20.83ns.

```

void OS_Launch(uint32_t theTimeSlice){
    STCTRL = 0; // disable SysTick during setup
    STCURRENT = 0; // any write to current clears it
    SYSPRI3 =(SYSPRI3&0x00FFFFFF)|0xE0000000; // priority 7
    STRELOAD = theTimeSlice - 1; // reload value
    STCTRL = 0x00000007; // enable, core clock and interrupt arm
    StartOS(); // start on the first task
}

```

*Program 3.7. RTOS launch.*

The **StartOS** is written in assembly (Program 3.8). In this simple implementation, the first user thread is launched by setting the stack pointer to the value of the first thread, then pulling all the registers off the stack explicitly. The stack is initially set up like it had been running previously, was interrupted (8 registers pushed), and then

suspended (another 8 registers pushed). When launch the first thread for the first time we do not execute a return from interrupt (we just pull 16 registers from its stack). Thus, the state of the thread is initialized and is now ready to run.

## StartOS

```
LDR R0, =RunPt ; currently running thread
LDR R1, [R0] ; R1 = value of RunPt
LDR SP, [R1] ; new thread SP; SP = RunPt->sp;
POP {R4-R11} ; restore regs r4-11
POP {R0-R3} ; restore regs r0-3
POP {R12}
ADD SP, SP, #4 ; discard LR from initial stack
POP {LR} ; start location
ADD SP, SP, #4 ; discard PSR
CPSIE I ; Enable interrupts at processor level
BX LR ; start first thread
```

*Program 3.8. Assembly code for the thread switcher.*

### 3.3.5. Switching threads

The SysTick ISR, written in assembly, performs the preemptive thread switch (Program 3.9). SysTick interrupts will be triggered at a fixed rate (e.g., every 2 ms in this example). Because SysTick is priority 7, it cannot preempt any background threads. This means SysTick can only suspend foreground threads. 1) The processor automatically saves eight registers (R0-R3, R12, LR, PC and PSR) on the stack as it suspends execution of the main program and launches the ISR. 2) Since the thread switcher has read-modify-write operations to the SP and to **RunPt**, we need to disable interrupts to make the ISR atomic. 3) Here we explicitly save the remaining registers (R4-R11). Notice the 16 registers on the stack match exactly the order of the 16 registers established by the **OS\_AddThreads** function. 4) Register R1 is loaded with **RunPt**, which points to the TCB of the thread in the process of being suspended. 5) By storing the actual SP into the sp field of the TCB, we have finished suspending the thread. To repeat, to suspend a thread we push all its registers on its stack and save its stack pointer in its TCB. 6) To implement round robin, we simply choose the next thread in the circular linked list and update **RunPt** with the new value. The #4 is used because the next field is the second entry in the TCB. We will change this step later to implement sleeping, blocking, and priority scheduling. 7) The first step of launching the new thread is to establish its stack pointer. 8) We explicitly pull eight registers from the stack. 9) We enable interrupts so the new thread runs with interrupts enabled. 10) The LR contains 0xFFFFFFF9 because a main program using MSP was suspended by SysTick. The BX LR instruction will automatically pull the remaining eight registers from the stack, and now the processor will be running the new thread.

The first time a thread runs, the only registers that must be set are PC, SP, the T-bit in the PSR (T=1), and the I-bit in the PSR (I=0). For debugging purposes, we do initialize the other registers the first time each thread is run, but these other initial values do not matter. We learned this trick of setting the initial register value to the register number (e.g., R5 is initially 0x05050505) from Micrium uC/OS-II. Notice in this simple example, the first time **Task0** runs it will be executed as a result of **StartOS**. However, the first time **Task1** and **Task2** are run, it will be executed as a result of running the **SysTick\_Handler**. In particular, the initial LR and PSR for **Task0** are set explicitly in **StartOS**, while the initial LR and PSR for Task1 and Task2 are defined in the initial stack set in **SetInitialStack**. An alternative approach to launching would have been to set the SP to the R4 field of its stack, set the LR to 0xFFFFFFFF9 and jump to line 8 of the scheduler. Most commercial RTOS use this alternative approach because it makes it easier to change. But we decided to present this StartOS because we feel it is easier to understand the steps needed to launch. Figure 3.12 shows three threads running in a round robin fashion.

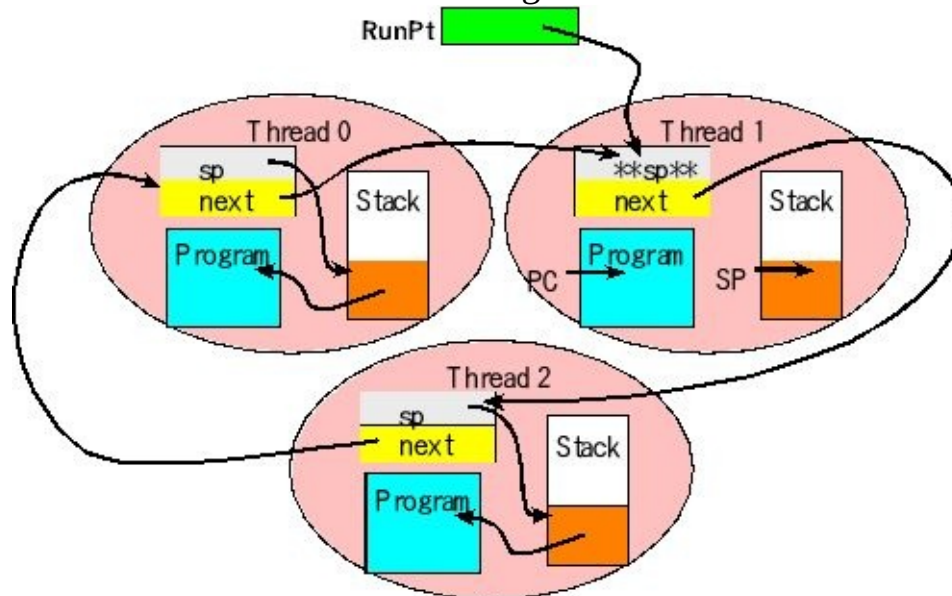


Figure 3.12. Three threads have their TCBs in a circular linked list. “\*\*\*sp\*\*\*” means this field is invalid for the one thread that is actually running.

```

SysTick_Handler          ; 1) Saves R0-R3,R12,LR,PC,PSR
CPSID  I                ; 2) Prevent interrupt during switch
PUSH  {R4-R11}          ; 3) Save remaining regs r4-11
LDR   R0, =RunPt        ; 4) R0=pointer to RunPt, old thread
LDR   R1, [R0]           ; R1 = RunPt
STR   SP, [R1]          ; 5) Save SP into TCB
LDR   R1, [R1,#4]       ; 6) R1 = RunPt->next
STR   R1, [R0]          ; RunPt = R1
LDR   SP, [R1]          ; 7) new thread SP; SP = RunPt->sp;
POP   {R4-R11}          ; 8) restore regs r4-11
CPSIE  I                ; 9) tasks run with interrupts enabled

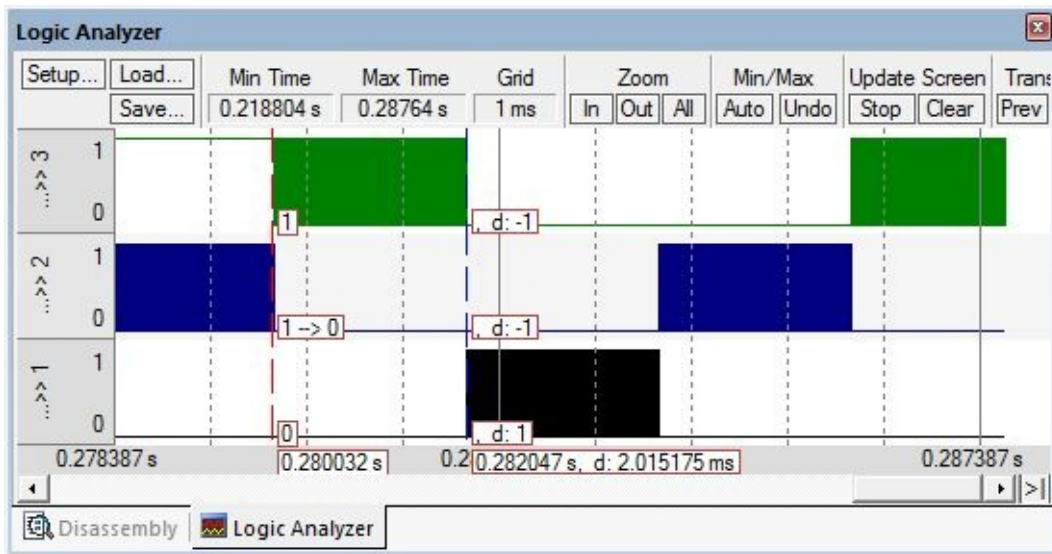
```

**BX LR ; 10) restore R0-R3,R12,LR,PC,PSR**

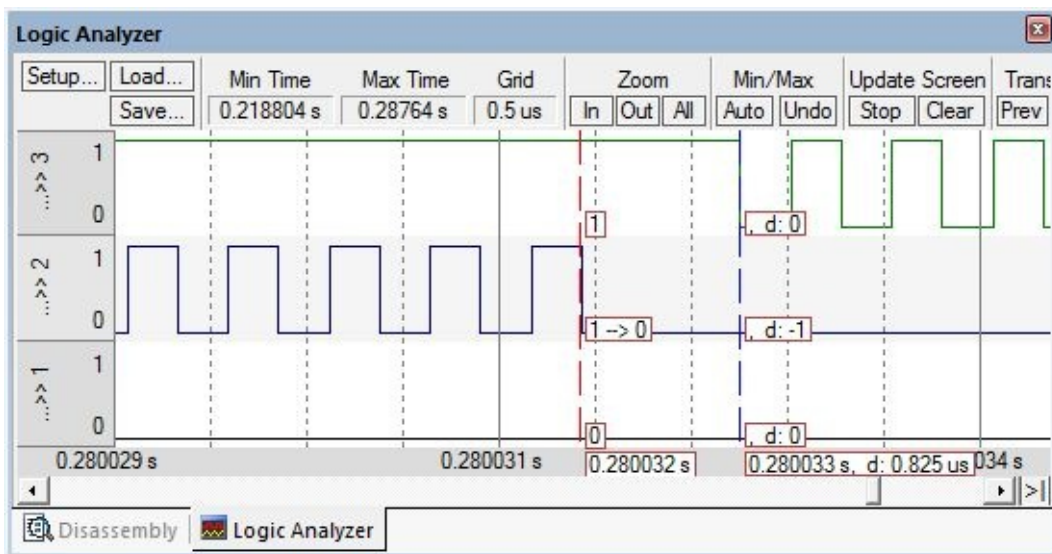
*Program 3.9. Assembly code for the thread switcher.*

### 3.3.6. Profiling the OS

You can find this simple RTOS in the starter projects as **RTOS\_xxx**, where xxx refers to the specific microcontroller on which the example was tested. Figures 3.13 and 3.14 show profiles of this RTOS at different time scales. We can estimate the thread switch time to be about 0.8  $\mu$ s, because of the gap between the last edge on one pin to the first edge on the next pin. In this case because the thread switch occurs every 2 ms, the 0.8- $\mu$ s thread-switch overhead is not significant.



*Figure 3.13. The RTOS runs three threads by giving each a 2ms, measured in simulator for the TM4C123.*



*Figure 3.14. Profile showing the thread switch time is about 0.8  $\mu$ s, measured in simulator for the TM4C123.*

### 3.3.7. Linking assembly to C

One of the limitations of the previous scheduler is that it's written entirely in assembly. Although fast, assembly programming is hard to extend and hard to debug. One simple way to extend this round robin scheduler is to have the assembly SysTick ISR call a C function, as shown in Program 3.10. The purpose of the C function is to run the scheduler and update the **RunPt** with the thread to run next. You can find this simple RTOS as **RoundRobin\_xxx**, where xxx refers to the specific microcontroller on which the example was tested.

```
void Scheduler(void){
    RunPt = RunPt->next; // Round Robin
}
```

*Program 3.10. Round robin scheduler written in C.*

The new SysTick ISR calls the C function in order to find the next thread to run, Program 3.11. We must save R0 and LR because these registers will not be preserved by the C function. **IMPORT** is an assembly pseudo-op to tell the assembler to find the address of **Scheduler** from the linker when all the files are being stitched together. Since this is an ISR, recall that LR contains 0xFFFFFFFF9, signifying we are running an ISR. We had to save the LR before calling the function because the BL instruction uses LR to save its return address. The POP instruction restores LR to 0xFFFFFFFF9. According to AAPCS, we need to push/pop an even number of registers (8-byte alignment) and functions are allowed to freely modify R0-R3, R12. For these two reasons, we also pushed and popped R0. Note that the other registers, R1,R2,R3 and R12 are of no consequence to us, so we don't bother saving them.

```
IMPORT Scheduler
SysTick_Handler      ; 1) Saves R0-R3,R12,LR,PC,PSR
    CPSID I          ; 2) Prevent interrupt during switch
    PUSH {R4-R11}    ; 3) Save remaining regs r4-11
    LDR R0, =RunPt    ; 4) R0=pointer to RunPt, old thread
    LDR R1, [R0]      ; R1 = RunPt
    STR SP, [R1]      ; 5) Save SP into TCB
; LDR R1, [R1,#4]    ; 6) R1 = RunPt->next
; STR R1, [R0]       ; RunPt = R1
    PUSH {R0,LR}
    BL Scheduler
    POP {R0,LR}
    LDR R1, [R0]      ; 6) R1 = RunPt, new thread
    LDR SP, [R1]      ; 7) new thread SP; SP = RunPt->sp;
    POP {R4-R11}     ; 8) restore regs r4-11
    CPSIE I          ; 9) tasks run with interrupts enabled
    BX LR            ; 10) restore R0-R3,R12,LR,PC,PSR
```

*Program 3.11. Assembly code for the thread switcher with call to the*

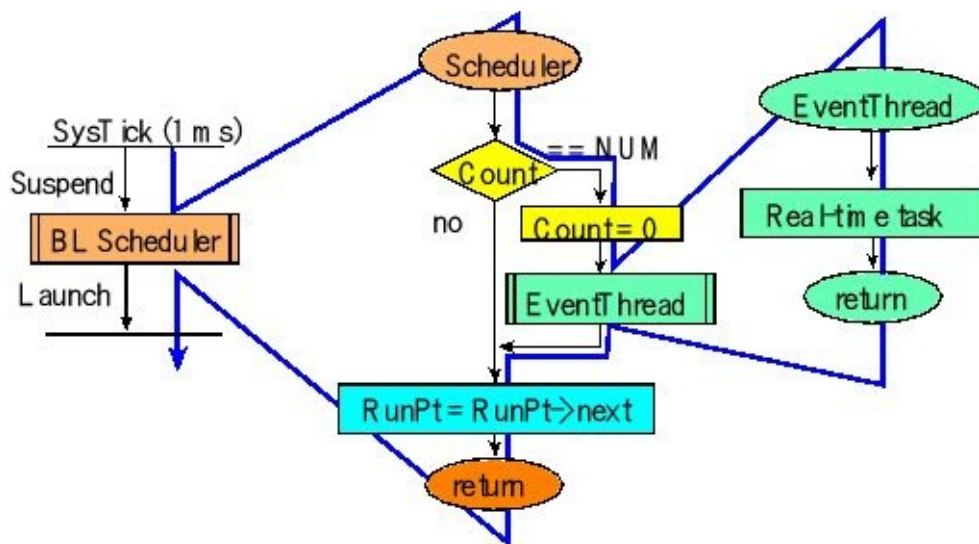
*scheduler written in C.*

In this implementation, we are running the C function **Scheduler** with interrupts disabled. On one hand this is good because all read-modify-write operations to shared globals will execute atomically, and not create critical sections. On the other hand, since interrupts are disabled, it will delay other possibly more important interrupts from being served. Running with interrupts disabled will cause time jitter for periodic threads and latency for event-response threads. A way to minimize jitter is to run the periodic tasks inside this Scheduler function itself.

### 3.3.8. Periodic tasks

A very appropriate feature of a RTOS is scheduling periodic tasks. If the number of periodic tasks is small, the OS can assign a unique periodic hardware timer for each task. Another simple solution is to run the periodic tasks in the scheduler. For example, assume the thread switch is occurring every 1 ms, and we wish to run the function **PeriodicUserTask()** every 10 ms, then we could modify the scheduler as shown in Figure 3.15 and Program 3.12. Assume the OS initialized the counter to 0. In order for this OS to run properly, the time to execute the periodic task must be very short and always return. These periodic tasks cannot spin or block.

This approach has very little time jitter because SysTick interrupts occur at a fixed and accurate rate. The SysTick ISR calls the Scheduler, and then the Scheduler calls the user task. The execution delay from the SysTick trigger to the running of the user task is a constant, so the time between executions of the user task is fixed and exactly equal to the SysTick trigger period.



*Figure 3.15. Simple mechanism to implement periodic event threads is to run them in the scheduler.*

```
uint32_t Counter;
```

```

#define NUM 10
void (*PeriodicTask1)(void); // pointer to user function
void Scheduler(void){
    if(++Counter == NUM){
        (*PeriodicTask1)(); // runs every NUM ms
        Counter = 0;
    }
    RunPt = RunPt->next; // Round Robin scheduler
}

```

*Program 3.12. Round robin scheduler with periodic tasks.*

If there are multiple real-time periodic tasks to run, then you should schedule at most one of them during each SysTick ISR execution. This way the time to execute one periodic task will not affect the time jitter of the other periodic tasks. For example, assume the thread switch is occurring every 1 ms, and we wish to run **PeriodicUserTask1()** every 10 ms, and run **PeriodicUserTask2()** every 25 ms. In this simple approach, the period of each task must be a multiple of the thread switch period. I.e., the periodic tasks must be multiples of 1 ms. First, we find the least common multiple of 10 and 25, which is 50. We let the counter run from 0 to 49, and schedule the two tasks at the desired rates, but at non-overlapping times as illustrated in Program 3.13.

```

uint32_t Counter;
void Scheduler(void){
    Counter = (Counter+1)%50; // 0 to 49
    if((Counter%10) == 1){ // 1, 11, 21, 31 and 41
        PeriodUserTask1();
    }
    if((Counter%25) == 0){ // 0 and 25
        PeriodUserTask2();
    }
    RunPt = RunPt->next; // Round Robin scheduler
}

```

*Program 3.13. Round robin scheduler with two periodic tasks.*

Consider a more difficult example, where we wish to run Task0 every 1 ms, Task1 every 1.5 ms and Task2 every 2 ms. In order to create non-overlapping executions, we will need a thread switch period faster than 1 kHz, so we don't have to run Task0 every interrupt. So, let's try working it out for 2 kHz, or 0.5 ms. The common multiple of 1, 1.5 and 2 is 6 ms. So we use a counter from 0 to 11, and try to schedule the three tasks. Start with Task0 running every other, and then try to schedule Task1 running every third. There is a conflict at 4 and 10.

Task0: runs every 1 ms at counter values 0, 2, 4, 6, 8, and 10

Task1: runs every 1.5 ms at counter values 1, 4, 7, and 10



So, let's try running faster at 4 kHz or every 0.25 ms. The common multiple is still 6 ms, but now the counter goes from 0 to 23. We can find a solution

Task0: runs every 1 ms at counter values 0, 4, 8, 12, 16, and 20

Task1: runs every 1.5 ms at counter values 1, 7, 13, and 19

Task2: runs every 2 ms at counter values 2, 10, and 18

In order this system to operate, the maximum time to execute each task must be very short compared to the period used to switch threads.



---

## 3.4. Semaphores

Remember that when an embedded system employs a real-time operating system to manage threads, typically this system combines multiple hardware/software objects to solve one dedicated problem. In other words, the components of an embedded system are tightly coupled. For example, in lab all threads together implement a personal fitness device. The fact that an embedded system has many components that combine to solve a single problem leads to the criteria that threads must have mechanisms to interact with each other. The fact that an embedded system may be deployed in safety-critical environments also implies that these interactions be effective and reliable.

We will use semaphores to implement synchronization, sharing and communication between threads. A **semaphore** is a counter with three functions: **OS\_InitSemaphore**, **OS\_Wait**, and **OS\_Signal**. Initialization occurs once at the start, but wait and signal are called at run time to provide synchronization between threads. Other names for wait are **pend** and **P** (derived from the Dutch word *proberen*, which means to test). Other names for signal are **post** and **V** (derived from the Dutch word *verhogen*, which means to increment).

The concept of a semaphore was originally conceived by the Dutch computer scientist Edsger Dijkstra in 1965. He received many awards including the 1972 Turing Award. He was the Schlumberger Centennial Chair of Computer Sciences at The University of Texas at Austin from 1984 until 2000. Interestingly he was one of the early critics of the GOTO instruction in high-level languages. Partly due to his passion, structured programming languages like C, C++ and Java have almost completely replaced non-structured languages like BASIC, COBOL, and FORTRAN.

In this book we will develop three implementations of semaphores, but we will begin with the simplest implementation called “spin-lock” (Figure 3.16). Each semaphore has a counter. If the thread calls **OS\_Wait** with the counter equal to zero it will “spin” (do nothing) until the counter goes above zero (Program 3.14). Once the counter is greater than zero, the counter is decremented, and the wait function returns. In this simple implementation, the **OS\_Signal** just increments the counter. In the context of the previous round robin scheduler, a thread that is “spinning” will perform no useful work, but eventually will be suspended by the SysTick handler, and then other threads will execute. It is important to allow interrupts to occur while the thread is spinning so that the software does not hang. The read-modify-write operations on the counter, `s`, is a critical section. So the read-modify-write sequence must be made atomic, because the scheduler might switch threads in between any two instructions that execute with the interrupts enabled. Program 3.14 shows the spinlock implementation of semaphores.

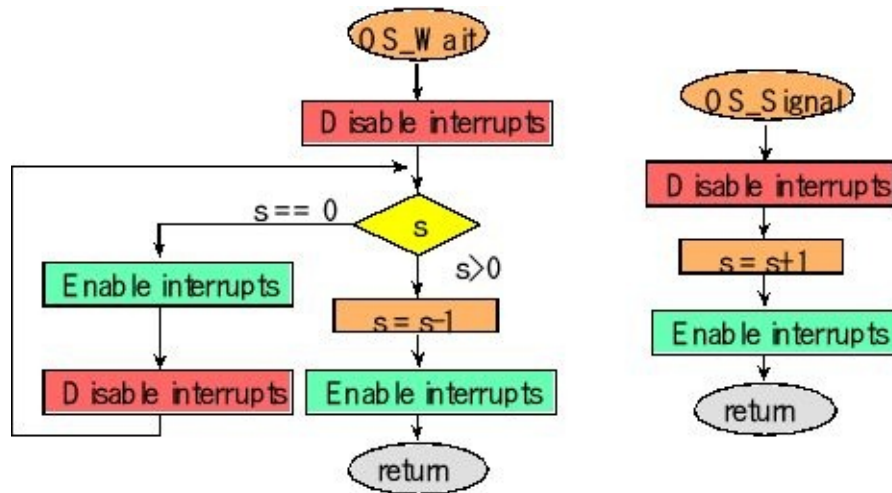


Figure 3.16. Flowcharts of a spinlock counting semaphore.

In the C implementation of spinlock semaphores, the tricky part is to guarantee all read-modify-write sequences are atomic. The while-loop reads the counter, which is always run with interrupts disabled. If the counter is greater than 0, it will decrement and store, such that the entire read-modify-write sequence is run with interrupts disabled. The while-loop must spend some time with interrupts enabled to allow other threads an opportunity to run, giving other threads an opportunity to call signal.

```

void OS_Wait(int32_t *s){
    DisableInterrupts();
    while((*s) == 0){
        ← EnableInterrupts(); // <- interrupts can occur here
        DisableInterrupts();
    }
    (*s) = (*s) - 1;
    EnableInterrupts();
}
void OS_Signal(int32_t *s){
    DisableInterrupts();
    (*s) = (*s) + 1;
    EnableInterrupts();
}

```

Program 3.14. A spinlock counting semaphore.

**Checkpoint 3.9:** What happens if we remove just the **EnableInterrupts** **DisableInterrupts** operations from while-loop of the spinlock **OS\_Wait** ?

**Checkpoint 3.10:** What happens if we remove all the **DisableInterrupts** **EnableInterrupts** operations from the spinlock **OS\_Wait** ?

In Program 3.15, Register R0 points to the semaphore counter. The LDREX

STREXcombination is a read-modify-write sequence that implements mutual exclusion. During a potential race condition, the first thread to execute LDREX captures exclusive access to the counter. When the thread with exclusive access performs STREX then the actual store will occur, and then the counter is considered free again. If a second thread executes LDREX during the period of exclusive access of another thread, it will capture an invalid version of the counter. However, when this second thread attempts STREX, it will not store. In this case, the assembly instruction **STREXPL R2,R1,[R0]** attempts to store the value in R1 through the pointer in R0. R2 is loaded with 0 if the store was allowed because this thread had exclusive access. On the other hand, R2 is loaded with 1 if the store did not happen because another thread had ownership. In this example, if R2 is nonzero, it will try it again.

```

OS_Wait      ;R0 points to counter
LDREX R1, [R0] ; counter
SUBS R1, #1 ; counter -1,
ITT PL ; ok if >= 0
STREXPL R2,R1,[R0] ; try update
CMPPL R2, #0 ; succeed?
BNE OS_Wait ; no, try again
BX LR

OS_Signal ; R0 points to counter
LDREX R1, [R0] ; counter
ADD R1, #1 ; counter + 1
STREX R2,R1,[R0] ; try update
CMP R2, #0 ; succeed?
BNE OS_Signal ;no, try again
BX LR

```

*Program 3.15. A spinlock counting semaphore that does not require disabling interrupts.*

**Observation:** If the semaphores can be implemented without disabling interrupts, then the latency in response to external events will be improved.

Spinlock semaphores are inefficient, wasting processor time when they spin on a counter with a value of zero. In the subsequent chapters we will develop more complicated schemes, like cooperation and blocking, to recover this lost time.


---

## 3.5. Thread Synchronization

### 3.5.1. Resource sharing, nonreentrant code or mutual exclusion

This section can be used in two ways. First it provides a short introduction to the kinds of problems that can be solved using semaphores. In other words, if you have a problem similar to one of these examples, then you should consider a thread scheduler with semaphores as one possible implementation. Second, this section provides the basic approach to solving these particular problems.

When we use a semaphore, we usually can assign a meaning or significance to the counter value. In the first application we could use a semaphore as a **lock** so only one thread at a time has access to a shared object. Another name for this semaphore is **mutex**, because it provides mutual exclusion. If the semaphore is 1 it means the object is free. If the semaphore is 0 it means the object is busy being used by another thread. For this application the initial value of the semaphore ( **x** ) is 1, because the object is initially free. A thread calls **OS\_Wait** to capture the object (decrement counter) and that same thread calls **OS\_Signal** to release the object (increment counter).

<pre>void Thread1(void){   Init1();   while(1){     OS_Wait(&amp;x);      // exclusive access      OS_Signal(&amp;x);     // other processing   } }</pre>	<pre>void Thread2(void){   Init2();   while(1){     OS_Wait(&amp;x);     // exclusive access      OS_Signal(&amp;x);     // other processing   } }</pre>
--	--

The objective of this example is to share a common resource on a one at a time basis, also referred to as “mutually exclusive” fashion. The critical section (or vulnerable window) of nonreentrant software is that region that should only be executed by one thread at a time. As an example, the common resource we will consider is a display device (LCD). Mutual exclusion in this context means that once a thread has begun executing a set of LCD functions, then no other thread is allowed to use the LCD. See Program 3.16. In other words, whichever thread starts to output to the LCD first will be allowed to finish outputting. The thread that arrives second will simply wait for the first to finish. Both will be allowed to output to the LCD, however, they will do

so on a one at a time basis. The mechanism to create mutual exclusion is to initialize the semaphore to 1, execute **OS\_Wait** at the start of the critical section, and then execute **OS\_Signal** at the end of the critical section. In this way, the information sent to one part of the LCD is not mixed with information sent to another part of the LCD.

Initially, the semaphore is 1. If **LCDmutex** is 1, it means the LCD is free. If **LCDmutex** is 0, it means the LCD is busy and no thread is waiting. In this chapter, a thread that calls **OS\_Wait** on a semaphore already 0 will wait until the semaphore becomes greater than 0. For a spinlock semaphore in this application, the possible values are only 0 (busy) or 1 (free). A semaphore that can only be 0 or 1 is called a **binary semaphore**.

```
void Task5(void){
    Init5();
    while(1){
        Unrelated5();
        OS_Wait(&LCDmutex);
        BSP_LCD_SetCursor(5, 0);
        BSP_LCD_OutUDec4(Time/10,COLOR);
        BSP_LCD_SetCursor(5, 1);
        BSP_LCD_OutUDec4(Steps,COLOR);
        BSP_LCD_SetCursor(16, 0);
        BSP_LCD_OutUFix2_1(TempData,COLOR);
        BSP_LCD_SetCursor(16, 1);
        BSP_LCD_OutUDec4(SoundRMS,COLOR);
        OS_Signal(&LCDmutex);
    }
}


void Task2(void){
    Init2();
    while(1){
        Unrelated2();
        OS_Wait(&LCDmutex);
        BSP_LCD_PlotPoint(Data,COLOR);
        BSP_LCD_PlotIncrement();
        OS_Signal(&LCDmutex);
    }
}
```

*Program 3.16. Semaphores used to implement mutual exclusion.*

### 3.5.2. Condition variable

In second application we could use a semaphore for synchronization. One example of this synchronization is a **condition variable**. If the semaphore is 0 it means an event has not yet happened, or things are not yet ok. If the semaphore is 1 it means the event has occurred and things are ok. For this application the initial value of the semaphore is 0, because the event is yet to occur. A thread calls **OS\_Wait** to wait for the event (decrement counter) and another thread calls **OS\_Signal** to signal that the event has occurred (increment counter). Let **event** be a semaphore with initial value of 0.

<b>void Thread1(void){</b>	<b>void Thread2(void){</b>
----------------------------	----------------------------

<pre> <b>Init1();</b>  <b>OS_Wait(&amp;event);</b> <b>// wait for event to occur</b> <b>while(1){</b>   <b>// other processing</b> <b>}</b> <b>}</b> </pre>	<pre> <b>Init2();</b> <b>// the event has occurred</b> <b>OS_Signal(&amp;event);</b> <b>while(1){</b>   <b>// other processing</b> <b>}</b> <b>}</b> </pre>
--	---

### 3.5.3. Thread communication between two threads using a mailbox

The objective of this example is to communicate between two main threads using a mailbox. In this first implementation both the producer and consumer are main threads, which are scheduled by the round robin thread scheduler (Program 3.17). The producer first generates data, and then it calls **SendMail ()**. Consumer first calls **RecvMail ()**, and then it processes the data. **Mail** is a shared global variable that is written by a producer thread and read by a consumer thread. In this way, data flows from the producer to the consumer. The **Send** semaphore allows the producer to tell the consumer that new mail is available. The **Ack** semaphore is a mechanism for the consumer to tell the producer, the mail was received. If **Send** is 0, it means the shared global *does not have* valid data. If **Send** is 1, it means the shared global *does have* valid data. If **Ack** is 0, it means the consumer *has not yet read* the global. If **Ack** is 1, it means the *consumer has read* the global. The sequence of operation depends on which thread arrives first. Initially, semaphores **Send** and **Ack** are both 0. Consider the case where the producer executes first.

<u>Execution</u>	<u>Mail</u>	<u>Send</u>	<u>Ack</u>	<u>Comments</u>
Initially	none	0	0	
Producer sets Mail	valid	0	0	Producer gets here first
Producer signals Send	valid	1	0	
Producer waits on Ack	valid	1	0	Producer spins because Ack =0
Consumer waits on Send	valid	0	0	Returns immediately because Send was 1
Consumer reads Mail	none	0	0	Reading once means Mail not valid
Consumer signals Ack	none	0	1	Consumer continues to execute
Producer finishes wait	none	0	0	Producer continues to execute

Next, consider the case where the consumer executes first.

<u>Execution</u>	<u>Mail</u>	<u>Send</u>	<u>Ack</u>	<u>Comments</u>
Initially	none	0	0	

Consumer waits on send	none	0	0	Consumer spins because Send =0
Producer sets Mail valid	0	0		Producer gets here second
Producer signals Send	valid	1	0	
Producer waits on Ack	valid	1	0	Producer spins because Ack =0
Consumer finishes wait	valid	0	0	Consumer continues to execute
Consumer reads Mail	none	0	0	Reading once means Mail not valid
Consumer signals Ack	none	0	1	Consumer continues to execute
Producer finishes wait	none	0	0	Producer continues to execute

There are two semaphores and one shared global data.

```
uint32_t Mail; // shared data
int32_t Send=0; // semaphore
int32_t Ack=0; // semaphore
```

The basic idea of this example is for one thread to send data to another. The producer calls **SendMail** and the consumer calls **RecvMail**.

<pre>void SendMail(uint32_t data){     Mail = data;     OS_Signal(&amp;Send);     OS_Wait(&amp;Ack); } void Producer(void){     Init1();     while(1){ uint32_t int myData;         myData = MakeData();         SendMail(myData);         Unrelated1();     } }</pre>	<pre>uint32_t RecvMail(void){     uint32_t theData;     OS_Wait(&amp;Send);     theData = Mail; // read mail     OS_Signal(&amp;Ack);     return theData; } void Consumer(void){     Init2();     while(1){ uint32_t thisData;         thisData = RecvMail();         Unrelated2();     } }</pre>
--	---

*Program 3.17. Semaphores used to implement a mailbox. Both Producer and Consumer are main threads.*

Remember that only main threads can call **OS\_Wait**, so the above implementation works only if both the producer and consumer are main threads.

If producer is an event thread, it cannot call **OS\_Wait**. For this scenario, we must remove the **Ack** semaphore and only use the **Send** semaphore (Program 3.18). Initially, the **Send** semaphore is 0. If **Send** is already 1 at the beginning of the

producer, it means there is already unread data in the mailbox. In this situation, data will be lost. In this implementation, the error count, **Lost**, is incremented every time the producer calls **SendMail()** whenever the mailbox is already full.

<pre> uint32_t Lost=0;  void SendMail(uint32_t data){     Mail = data;     if(Send){         Lost++;     }else{         OS_Signal(&amp;Send);     } } void Producer(void){     uint32_t int myData;     myData = MakeData();     SendMail(myData);     Unrelated1(); } </pre>	<pre> uint32_t RecvMail(void){     OS_Wait(&amp;Send);     return Mail; // read mail }  void Consumer(void){     Init2();     while(1){ uint32_t thisData;         thisData = RecvMail();         Unrelated2();     } } </pre>
---	--

*Program 3.18. Semaphores used to implement a mailbox. Producer is an event thread and Consumer is a main thread.*

**Checkpoint 3.11:** There are many possible ways to handle the case where data is lost in Program 3.18. The code as written will destroy the old data, and the consumer will skip processing the old lost data. Modify Program 3.18 such that the system destroys the new data, and the consumer will skip processing the new data.

A mailbox forces the producer and consumer to execute lock-step {producer, consumer, producer, consumer,...}. It also suffers from the potential to lose data. Both of these limitations will motivate the **first in first out** (FIFO) queue presented in the next chapter.



## 3.6. Process Management

One of the requirements of a thread manager was that threads be **tightly coupled**, sharing a common objective. In this context, tightly coupled is categorized by threads that share global data and share I/O devices. However, if we have multiple software tasks that are **loosely coupled** then we require a more complex scheduler. Again in this context, loosely coupled means that they do not share data or I/O devices. We define **processes** as software tasks that are loosely coupled. Each process has its own stack, code, data (globals), and heap. The stack, code, data, and I/O of one process are not shared with other processes. See Figure 3.17.

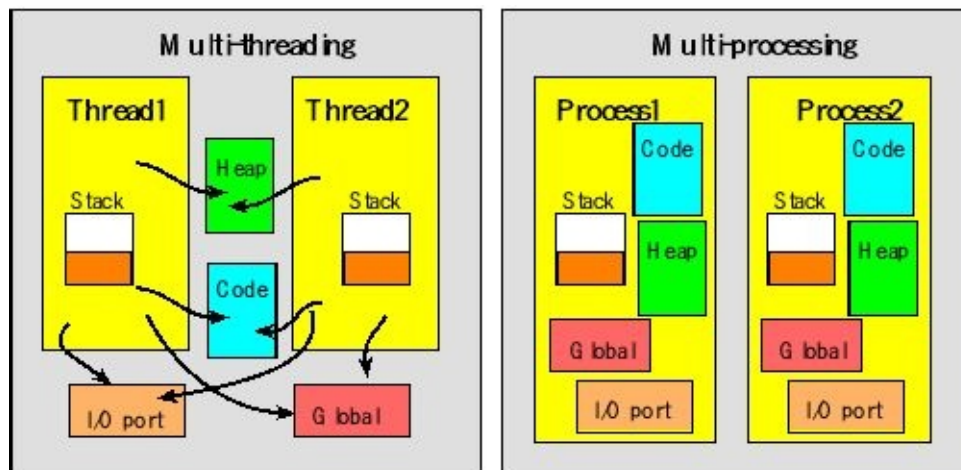


Figure 3.17. Comparison of threads and processes.

In Unix, a new process is created using the **fork()** command. To load and execute a process, the Unix command is **exec()**. An existing process can be initialized with the **init()** command. The function **exit()** will terminate the process, and the OS will recover all resources (memory, I/O). The function **exit()** is automatically called when the **main()** program returns. In Windows the **CreateProcess()** function will create a new process and load the program image. The function **ExitProcess()** will terminate the process and recover the resources.

The OS will provide mechanisms for the processes to communicate with each other, but in general, processes do not have a shared objective. The OS handles the loading of processes into memory. On a microcontroller, the memory image will have multiple segments: code, stack, heap and data.

On a more sophisticated processor, the OS will configure the memory protection unit to prevent one process from accessing the memory space of another, see Figure 3.18. The Cortex M microcontrollers do not have this type of memory protection.

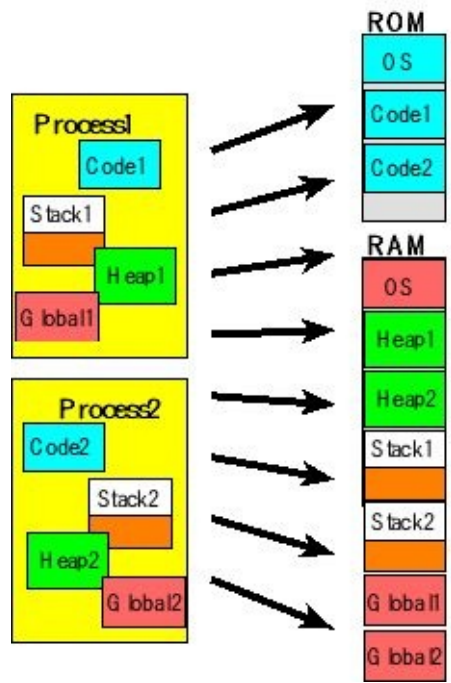


Figure 3.18. Loading processes into physical memory of a microcontroller.

## 3.7. Dynamic loading and linking

The **Executable and Linking Format** (ELF) standard simplifies software development by providing with a set of binary interface definitions that apply to multiple environments. For example, code created with one compiler can be combined with code created with a second compiler and these two software objects can be executed together. The standard reduces the need for compiling all software objects into one project.

After we compile and before we execute code onto a microcontroller, the various software modules must be combined (linked) and then loaded (programmed into ROM). The following lists some of the **sections** used by the Keil IDE

- Linking: sections
- Object files -> executables
- Code (RO / .text)
- Data (RW / .data)
- Zero data (ZI / .bss)
- String/symbol table

**Object files** are binary representations of software created by the compiler and linker that can be executed by the processor. For convenience there are two parallel views of the same object file. The linker interacts using the **Linking View** and the operating system interacts using the **Executable View** at run time, as shown in Figure 3.19.

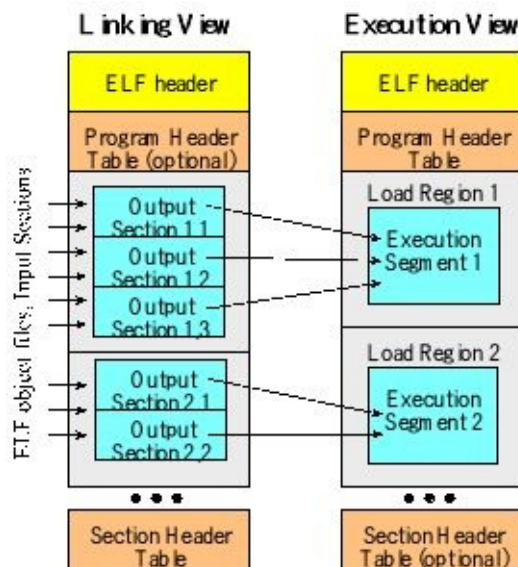


Figure 3.19. Object file format.

The ELF header describes the organization of the file. The sections contain object file information such as instructions, data, symbol table, and relocation information.

The program header table explains how to create the image. If the file is used to create an executable image, then it will have a program header table. Relocatable files do not need a program header table. The section header table describes the sections, including section name, size, and type. Files used during linking must have a section header table, and files used during execution need not have a section header table. The figure implies an order for sections and segments, however only the ELF header has a fixed position, while sections and segments have no order.

On a microcontroller like the Cortex M there are three types of memory segments. See Figure 3.20. Typically, we place instructions and fixed constants in ROM. Keil labels this segment as RO or .text. As we can see the compiler creates an image where this **RO segment** begins at 0x00000000. However, when this segment is loaded into memory, it is combined with other ROM segments and possibly moved to another position in ROM other than 0x00000000. The second type of segment is RW or .data segment. This segment contains global variables that have initial values. The compiler creates an image where this **RW segment** begins at 0x20000000. However, when this segment is loaded into memory, it is combined with other RW segments and possibly moved to another position in RAM other than 0x20000000. The **ZI segment** also contains global variables; however, these variables are initialized to 0. Again, when the ZI segments are loaded, these too may be moved to other positions in RAM. **Loading** is the process of placing all these segments into appropriate places in memory. **Linking** is defined as the process of combining the segments and fixing up all cross referenced addresses. The executable image also includes a starting location for execution.

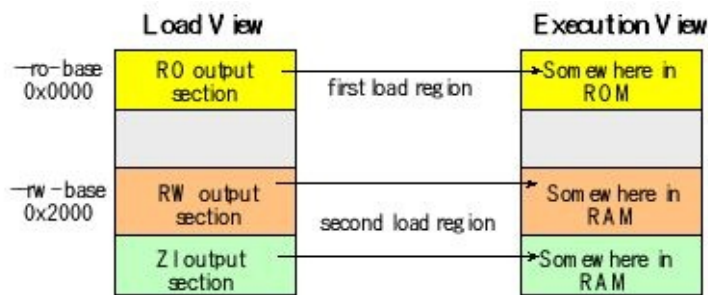


Figure 3.20. Loading and linking takes compiler output and makes it ready to run.

For simple projects, the entire compile, link and load operations occur statically when one issues a build/download command to the Keil IDE. For more complex projects, we could compile a process into its ELF format and then dynamically load/link the process at run time. To facilitate dynamic linking we compile the program into **position independent code** (PIC). Another name for object code that will run regardless of its position in memory is **relocatable code**. Most ARM object code is relocatable because the branch instructions use PC-relative addressing. Branches using the **BX** instruction will not be relocatable. Function pointers typically use the **BX** instruction, so they will be trickier to link.

Dynamic linking to global data can be achieved with a static base register, called

**position-independent data.** In the following example, R9 points to the beginning of the global variable space for this program. R9 is set dynamically by the loader/linker. All global variables have a fixed offset from this base register.

<pre>; regular access to global v1 SPACE 4 ; global  f1 LDR R1,=v1    LDR R2,[R1] ; contents of v1</pre>	<pre>uint32_t v2; // global ;R9 points to global space ;ofs is offset of this variable LDR R2,[R9,#ofs]</pre>
--	---

With dynamic linking and loading we need a mechanism to call OS functions that are not compiled with the user program. The typically solution for linking to OS functions is to use the **SVC** or software trap instruction. The implementation of software trap was described in Section 2.2.6.

For a detailed description of the ELF format, search “ELF” on <http://infocenter.arm.com>

---

## 3.8. Exercises

3.1 In 16 words or less for each, give definitions of the following terms: jitter, real-time, call back, profile, semaphore, and scheduler.

3.2 Compare and contrast thread and process.

3.3 Compare and contrast event thread and main thread.

3.4 Compare and contrast parallel, distributed and concurrent programming.

3.5 Compare and contrast hard, firm and soft real time. Give an example of each different from the ones in the chapter.

3.6 Please name the following schedulers (round robin, rate monotonic, priority, cooperative, and exponential queue):

A. A dynamic scheduler that shifts importance depending on if the thread ran to the completion of its time slice.

B. Run the ready threads in circular fashion, giving each the same amount of time to execute

C. Assign importance according to these periods with more frequent tasks having higher importance.

D. Threads themselves decide when to stop running

E. Run the most important ready threads first, running less important threads only if there are no important threads ready

3.7 We can use semaphores to limit access to resources. In the following example both threads need access to a printer and an SPI port. The binary semaphore **sPrint** provides mutual exclusive access to the printer and the binary semaphore **sSPI** provides mutual exclusive access to the SPI port. Consider the following scenario to see if it has any bugs.

<i>Thread 1</i> <b>bwait(&amp;sPrint);</b> <b>bwait(&amp;sSPI);</b> <b>OutSPI(4);</b> <b>printf("Hasta luego");</b> <b>OutSPI(6);</b> <b>bsignal(&amp;sPrint);</b> <b>bsignal(&amp;sSPI);</b>	<i>Thread 2</i> <b>bwait(&amp;sSPI);</b> <b>bwait(&amp;sPrint);</b> <b>OutSPI(5);</b> <b>printf("tchau");</b> <b>OutSPI(7);</b> <b>bsignal(&amp;sSPI);</b> <b>bsignal(&amp;sPrint);</b>
--	--

If there is a bug, show the correction

3.8 You have three tasks. Task 1 takes a maximum of 1 ms to execute and runs every 10 ms. Task 2 takes a maximum of 0.5 ms to execute and runs every 1 ms. Task 3

takes a maximum of 1 ms to execute and runs every 100 ms. Is there a possible scheduling algorithm for these three tasks?

**3.9** You have four tasks. Task 1 takes a maximum of 1 ms to execute and runs every 5 ms. Task 2 takes a maximum of 0.5 ms to execute and runs every 2 ms. Task 3 takes a maximum of 1 ms to execute and runs every 20 ms. Task 4 takes a maximum of 6 ms to execute and runs every 10 ms. Is there a possible scheduling algorithm for these three tasks?

**3.10** You have four tasks. Task 1 takes a maximum of 1 ms to execute and runs every 5 ms. Task 2 takes a maximum of 0.5 ms to execute and runs every 2 ms. Task 3 takes a maximum of 1 ms to execute and runs every 20 ms. Task 4 takes a maximum of 5 ms to execute and runs every 10 ms. Do you think a scheduling algorithm exists? Justify your answer.

# 4. Time Management

**Chapter 4 objectives are to:**

- Implement cooperation using OS\_Suspend
- Design and implement blocking semaphores
- Implement data flow with first in first out (FIFO) queues
- Implement sleeping
- Employ periodic interrupts to manage periodic tasks

An important aspect of real-time systems is managing time, more specifically minimizing wastage of time through an idle busy-wait. Such busy-wait operations were used in our simple implementation of semaphores in the last chapter. In this chapter we will see how we can recover this wasted time.



## 4.1. Cooperation

### 4.1.1. Spin-lock semaphore implementation with cooperation

Sometimes the OS or the thread knows the thread can no longer make progress. If a thread wishes to cooperatively release control of the processor it can call **OS\_Suspend**, which will halt this thread and run another thread. Because all the threads work together to solve a single problem, adding cooperation at strategic places allows the system designer to greatly improve performance. When threads wish to suspend themselves, they call **OS\_Suspend**. Again, the SysTick ISR must be configured as a priority 7 interrupt so that it does not attempt to suspend any hardware ISRs that may be running. **OS\_Suspend** can only be called by a main thread. Note that it is possible to force a SysTick interrupt by bypassing the normal “count to zero” event that causes it. To do this, we write a 1 to bit 26 of the **INTCTRL** register, which causes the SysTick interrupt. Writing zeros to the other bits of this register has no effect. This operation will set the **Countflag** in SysTick and the ISR will suspend the current thread, runs the SysTick\_Handler (which calls the scheduler), and then launch another thread. In this first implementation, we will not reset the SysTick timer from interrupting normally (count to zero). Rather we simply inject another execution of the ISR. If we were 75% through the 1-ms time slice when **OS\_Suspend** is called, this operation will suspend the current thread and grant the remaining 0.25-ms time to the next thread.

One way to make a spin-lock semaphore more efficient is to place a suspend in the while loop as it is spinning, as shown on the right of Figure 4.1 and as Program 4.1. This way, if the semaphore is not available, the thread stops running. If there are  $n$  other running threads and the time slice is  $\Delta t$ , then the semaphore is checked every  $n*\Delta t$ , and very little processor time is wasted on the thread which cannot run. One way to suspend a thread is to trigger a SysTick interrupt.

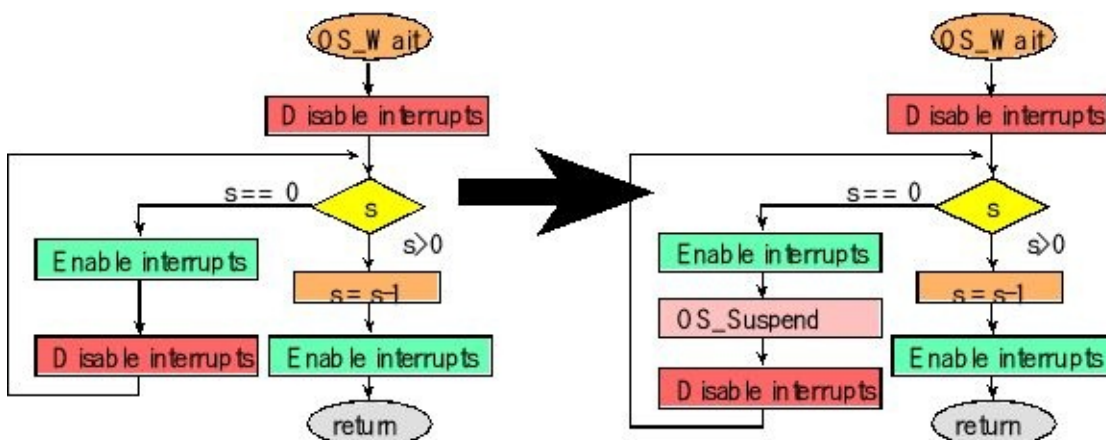


Figure 4.1. Regular and efficient implementations of spinlock wait.

```

void OS_Suspend(void){
    INTCTRL = 0x04000000; // trigger SysTick
}
void OS_Wait(int32_t *s){
    DisableInterrupts();
    while((*s) == 0){
        EnableInterrupts();
        OS_Suspend(); // run thread switcher
        DisableInterrupts();
    }
    (*s) = (*s) - 1;
    EnableInterrupts();
}

```

*Program 4.1. A cooperative thread switch will occur if the software explicitly triggers a thread switch.*

**Checkpoint 4.1:** Assume the thread scheduler switches threads every 1 ms without cooperation, and there are 5 total threads in the scheduler. If a thread needs to wait 1 second for its semaphore to be incremented, how much time will spinlock implementation waste, spinning in **OS\_Wait** doing no useful work?

**Checkpoint 4.2:** Assume the thread scheduler switches threads every 1 ms, one thread is spinning in **OS\_Wait** because its semaphore is 0, and there are 4 other running threads that are not spinning. Assuming **OS\_Wait** is implemented like Program 4.1 with cooperation, how often is the loop in **OS\_Wait** run?

The implementation in Program 4.1 did not reset the SysTick counter on a cooperative thread switch. So it is unfair for the thread that happens to be run next. However, in this implementation, since SysTick interrupts are still triggered every 1 ms, SysTick can be used to perform periodic tasks. Once we shift the running of periodic tasks to another timer ISR, we will be able to use this fair implementation of suspend:

```

void OS_Suspend(void){
    STCURRENT = 0; // reset counter
    INTCTRL = 0x04000000; // trigger SysTick
}

```

Using this version of suspend, if we are 75% through the 1-ms time slice when **OS\_Suspend** is called, this operation will suspend the current thread and grant a full 1-ms time to the next thread. We will be able to use this version of suspend once we move the periodic event threads away from SysTick and onto another timer interrupt.

One way to handle periodic event threads is to use a separate periodic interrupt (not the same SysTick that is used for thread switching.) This means the accurate running of event threads will not be disturbed by resetting the SysTick timer. Although you

could use either version of **OS\_Suspend** , resetting the counter will be fairer.

## 4.1.2. Cooperative Scheduler

In this section we will develop a 3-thread cooperative round-robin scheduler by letting the tasks suspend themselves by triggering a SysTick interrupt.

You can find this cooperative OS as **Cooperative\_xxx**, where xxx refers to the specific microcontroller on which the example was tested, Program 4.2. Figure 4.2 shows a profile of this OS. We can estimate the thread switch time to be about 1  $\mu$ s, because of the gap between the last edge on one pin to the first edge on the next pin. In this case, because the thread switch occurs every 1.3  $\mu$ s, the 1- $\mu$ s thread-switch overhead is significant. Even though SysTick interrupts are armed, the SysTick hardware never triggers an interrupt. Instead, each thread voluntarily suspends itself before the 1-ms interval.

```
void Task0(void){
    Count0 = 0;
    while(1){
        Count0++;
        Profile_Toggle0(); // toggle bit
        OS_Suspend();
    }
}
void Task1(void){
    Count1 = 0;
    while(1){
        Count1++;
        Profile_Toggle1(); // toggle bit
        OS_Suspend();
    }
}
void Task2(void){
    Count2 = 0;
    while(1){
        Count2++;
        Profile_Toggle2(); // toggle bit
        OS_Suspend();
    }
}
```

*Program 4.2. User threads that use a cooperative scheduler.*



*Figure 4.2. The OS runs three threads; each thread volunteers to suspend running in simulation mode on the TM4C123. The three profile pins from Program 4.2 are measured versus time using a logic analyzer.*

We must use a separate periodic interrupt to manage periodic tasks when running a cooperative scheduler, so that the timing of periodic events would be regular.

---

## 4.2. Blocking semaphores

### 4.2.1. The need for blocking

The basic idea of a **blocking semaphore** will be to prevent a thread from running (we say the thread is blocked) when the thread needs a resource that is unavailable. There are three reasons we will replace spin-lock semaphores with blocking semaphores. The first reason is an obvious **inefficiency** in having threads spin while there is nothing for them to do. Blocking semaphores will be a means to recapture this lost processing time. Essentially, with blocking semaphores, a thread will not run unless it has useful work it can accomplish. Even with spinlock/cooperation it is wasteful to launch a thread you know can't run, only to suspend itself 10  $\mu$ s later.

The second problem with spin-lock semaphores is a **fairness** issue. Consider the case with threads 1 2 3 running in round robin order. Assume thread 1 is the one calling **Signal**, and threads 2 and 3 call **Wait**. If threads 2 and 3 are both spinning waiting on the semaphore, and then thread 1 signals the semaphore, which thread (2 or 3) will be allowed to run? Because of its position in the 1 2 3 cycle, thread 2 will always capture the semaphore ahead of thread 3. It seems fair when the status of a resource goes from busy to available, that all threads waiting on the resource get equal chance. A similar problem exists in non-computing scenarios where fairness is achieved by issuing numbered tickets, creating queues, or having the customers sign a log when they enter the business looking for service. E.g., when waiting for a checkout clerk at the grocery store, we know to get in line, and we think it is unfair for pushy people to cut in line in front of us. We define **bounded waiting** as the condition where once a thread begins to wait on a resource (the call to **OS\_Wait** does not return right away), there are a finite number of threads that will be allowed to proceed before this thread is allowed to proceed. Bounded waiting does not guarantee a minimum time before **OS\_Wait** will return; it just guarantees a finite number of other threads will go before this thread. For example, it is holiday time, I want to mail a package to my mom, I walk into the post office and take a number, the number on the ticket is 251, I look up at the counter and the display shows 212, and I know there are 39 people ahead of me in line. We could implement bounded waiting with blocking semaphores by placing the blocked threads on a list, which is sorted by the order in which they blocked. When we wake up a thread off the blocked list, we wake up the one that has been waiting the longest. We introduce the concept of bounded waiting because it is a feature available in most commercial operating systems.

The third reason to develop blocking semaphores will be the desire to implement a **priority thread scheduler**. With a round-robin scheduler we assume each thread has equal importance. With a priority scheduler we will run the highest priority thread

that is ready to run. For example, if we have one high priority thread that is ready, we will run it over and over regardless of whether or not there are any lower priority threads ready. We will discuss the issues of starvation, aging, inversion and inheritance in the next chapter. A priority scheduler will require the use of blocking semaphores. I.e., we cannot use a priority scheduler with spin-lock semaphores.

## 4.2.2. The blocked state

A thread is in the **blocked state** when it is waiting for some external event like input/output (keyboard input available, printer ready, I/O device available.) We will use semaphores to implement communication and synchronization, and it is semaphore function **OS\_Wait** that will block a thread if it needs to wait. For example, if a thread communicates with other threads then it can be blocked waiting for an input message or waiting for another thread to be ready to accept its output message. If a thread wishes to output to the display, but another thread is currently outputting, then it will block. If a thread needs information from a FIFO (calls **Get**), then it will be blocked if the FIFO is empty (because it cannot retrieve any information.) Also, if a thread outputs information to a FIFO (calls **Put**), then it will be blocked if the FIFO is full (because it cannot save its information.) The semaphore function **OS\_Signal** will be called when it is appropriate for the blocked thread to continue. For example, if a thread is blocked because it wanted to print and the printer was busy, it will be signaled when the printer is free. If a thread is blocked waiting on a message, it will be signaled when a message is available. Similarly, if a thread is blocked waiting on an empty FIFO, it will be signaled when new data are put into the FIFO. If a thread is blocked because it wanted to put into a FIFO and the FIFO was full, it will be signaled when another thread calls **Get**, freeing up space in the FIFO.

Figure 4.3 shows five threads. In this simple implementation of blocking we add a third field, called **blocked**, to the TCB structure, defining the status of the thread. The **RunPt** points to the TCB of the thread that is currently running. The **next** field is a pointer chaining all five TCBs into a circular linked list. Each TCB has a **StackPt** field. Recall that, if the thread is running it is using the real SP for its stack pointer. However, the other threads have their stack pointers saved in this field. The third field is a **blocked** field. If the **blocked** field is null, there are no resources preventing the thread from running. On the other hand, if a thread is blocked, the **blocked** field contains a pointer to the semaphore on which this thread is blocked. In Figure 4.3, we see threads 2 and 4 are blocked waiting for the resource (semaphore **free**). All five threads are in the circular linked list although only three of them will be run.

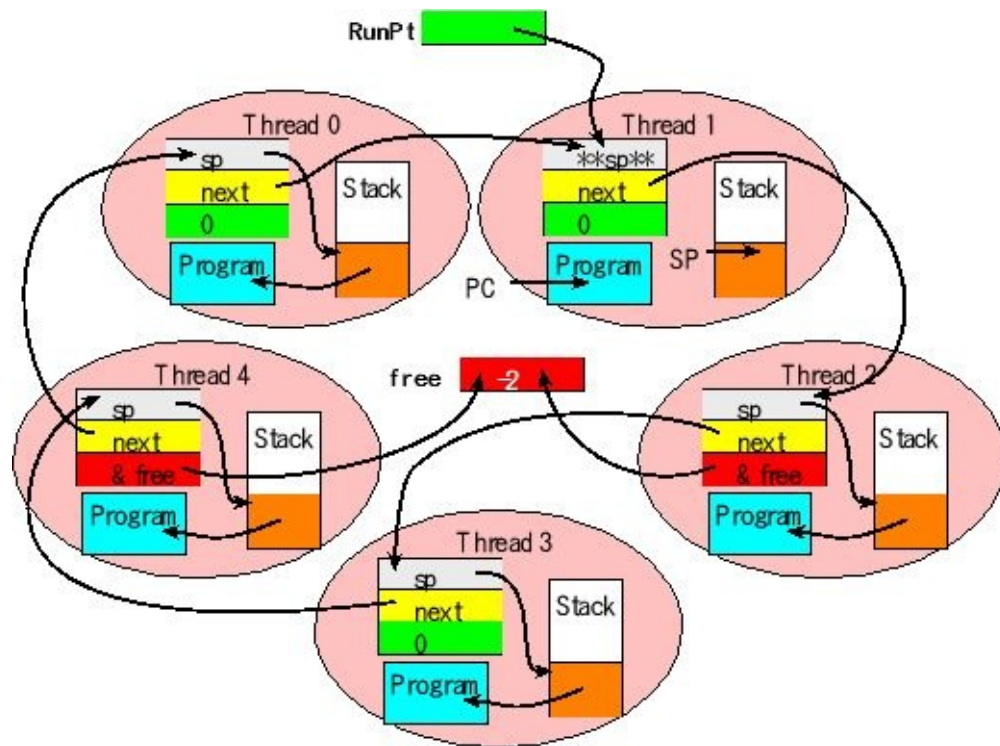


Figure 4.3. Threads 0, 1 and 3 are being run by the scheduler. Threads 2 and 4 are blocked on **free** and will not run until some thread signals **free**.

In this simple approach, a main thread can only be blocked on one resource. In other words, when a thread calls **OS\_Wait** on a semaphore with value 0 or less, that thread is blocked and stops running. Therefore, once blocked on one semaphore, it cannot block on a second semaphore. Figure 4.3 shows just one semaphore, but even when there are multiple semaphores, we need only one **blocked** field in the TCB. Since C considers zero as false and nonzero as true, the **blocked** field can also be considered as a Boolean, specifying whether or not the thread is blocked. This simple solution is adequate for systems with a small number of threads (e.g., less than 20).

Notice in this simple implementation we do not maintain a separate linked list of threads blocked on a specific semaphore. In particular, in Figure 4.3 we know threads 2 and 5 are blocked on the semaphore **free**, but we do not know which thread blocked first. The advantage of this implementation using one circular linked list data structure to hold the TCBs of all the threads will be speed and simplicity. Note that, we need to add threads to the TCB list only when created, and remove them from the TCB list if the thread kills itself. If a thread cannot run (blocked) we can signify this event by setting its **blocked** field like Figure 4.3 to point to the semaphore on which the thread is blocked.

In order to implement bounded waiting, we would have to create a separate blocked linked list for each reason why the thread cannot execute. For example, we could have one blocked list for threads waiting for the output display to be free, one list for threads waiting because a FIFO is full, and one lists for threads waiting because another FIFO is empty. In general, we will have one blocked list with each reason a

thread might not be able to run. This approach will be efficient for systems with many threads (e.g., more than 20). These linked lists contain threads sorted in order of how long they have been waiting. To implement bounded waiting, when we signal a semaphore, we wake up the thread that has been waiting the longest.

In this more complex implementation, we unchain a TCB from the ready circular linked list when it is blocked. In this way a blocked thread will never run. We place the blocked TCBs on a linear linked list associated with the semaphore (the reason it was blocked). We can implement bounded waiting by putting blocked TCBs at the end of the list and waking up threads from the front of the list. There will be a separate linked list for every semaphore. This method is efficient when there are many threads that will be blocked at one time. The thread switching will be faster because the scheduler will only see threads that could run, and not have to look at blocked threads in the circular linked list. Most commercial operating systems implement blocking by unchaining blocked threads because they need to operate efficiently with dozens of threads.

However, for simple operating systems that manage less than 10 threads it will be faster and easier to not implement unchaining. Rather, simple schedulers can skip threads with a nonzero **blocked** field.

### 4.2.3. Implementation

We will present a simple approach for implementing blocking semaphores. Notice in Figure 4.4 that wait always decrements and signal always increments. This means the semaphore can become negative. In the example of using a semaphore to implement mutual exclusion, if **free** is 1, it means the resource is free. If **free** is 0, it means the resource is being used. If **free** is -1, it means one thread is using the resource and a second thread is blocked, waiting to use it. If **free** is -2, it means one thread is using the resource and two other threads are blocked, waiting to use it.

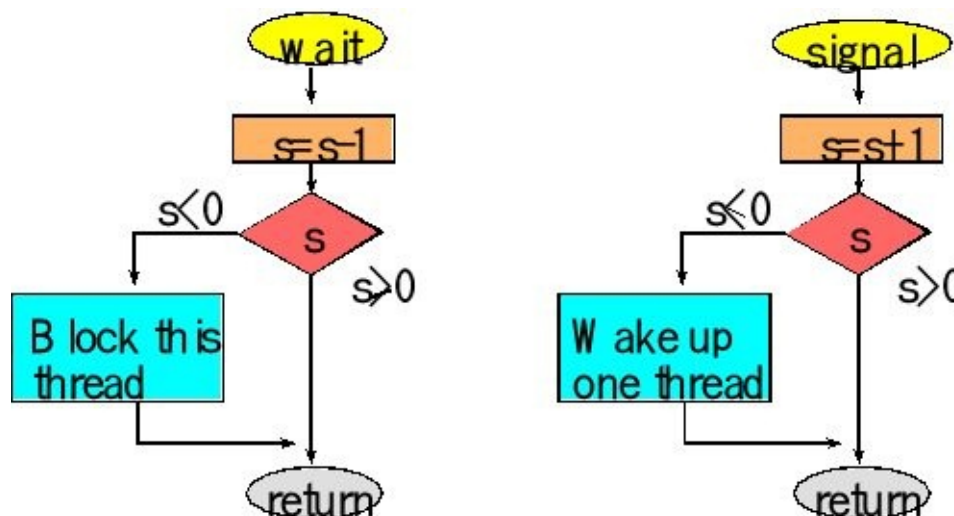


Figure 4.4. Flowcharts of a blocking counting semaphore.



In this simple implementation, the semaphore is a signed integer. This implementation of blocking is appropriate for systems with less than 20 threads. In this implementation, a **blocked** field is added to the TCB. The type of this field is a pointer to a semaphore. The semaphore itself remains a signed integer. If **blocked** is null, the thread is not blocked. If the **blocked** field contains a semaphore pointer, it is blocked on that semaphore. The “Block this thread” operation will set the **blocked** field to point to the semaphore, then suspend the thread.

```

void OS_Wait(int32_t *s){
    DisableInterrupts();
    (*s) = (*s) - 1;
    if((*s) < 0){
        RunPt->blocked = s; // reason it is blocked
        EnableInterrupts();
        OS_Suspend();    // run thread switcher
    }
    EnableInterrupts();
}

```

The “Wakeup one thread” operation will be to search all the TCBs for first one that has a **blocked** field equal to the semaphore and wake it up by setting its **blocked** field to zero

```

void OS_Signal(int32_t *s){
    tcbType *pt;
    DisableInterrupts();
    (*s) = (*s) + 1;
    if((*s) <= 0){
        pt = RunPt->next; // search for one blocked on this
        while(pt->blocked != s){
            pt = pt->next;
        }
        pt->blocked = 0; // wakeup this one
    }
    EnableInterrupts();
}

```

Notice in this implementation, calling the signal will not invoke a thread switch. During the thread switch, the OS searches the circular linked-list for a thread with a **blocked** field equal to zero (the woken up thread is a possible candidate). This simple implementation will not allow you to implement bounded waiting. *Notice that this solution does not implement bounded waiting.*

```

void Scheduler(void){
    RunPt = RunPt->next; // run next thread not blocked
    while(RunPt->blocked){ // skip if blocked

```

```

    RunPt = RunPt->next;
  }
}

```

**Checkpoint 4.3:** Assume the RTOS is running with a preemptive thread switch every 1 ms. Assume there are 8 threads in the TCB circular list, and 5 of the threads are blocked. Assume the while loop in the above **Scheduler** function takes 12 assembly instructions or 150ns to execute each time through the loop. What is the maximum time wasted in the scheduler looking at threads that are blocked? In other words, how much time could be saved by unchaining blocked threads from the TCB list?

## 4.2.4. Thread rendezvous

The objective of this example is to synchronize Threads 1 and 2 (Program 4.3). In other words, whichever thread gets to this part of the code first will wait for the other. Initially semaphores **S1** and **S2** are both 0. The two threads are said to **rendezvous** at the code following the signal and wait calls. The rendezvous will cause thread 1 to execute **Stuff1** at the same time (concurrently) as thread 2 executes its **Stuff2**.

<pre> void Task1(void){ // Thread 1   Init1();   while(1){     Unrelated1();     OS_Signal(&amp;S1);     OS_Wait(&amp;S2);     Stuff1();   } } </pre>	<pre> void Task2(void){ // Thread 2   Init2();   while(1){     Unrelated2();     OS_Signal(&amp;S2);     OS_Wait(&amp;S1);     Stuff2();   } } </pre>
---	---

*Program 4.3. Semaphores used to implement rendezvous.*

There are three scenarios the semaphores may experience and their significance is listed below:

S1	S2	Meaning
0	0	Neither thread has arrived at the rendezvous location or both have passed
-1	+1	Thread 2 arrived first and Thread 2 is blocked waiting for Thread 1
+1	-1	Thread 1 arrived first and Thread 1 is blocked waiting for

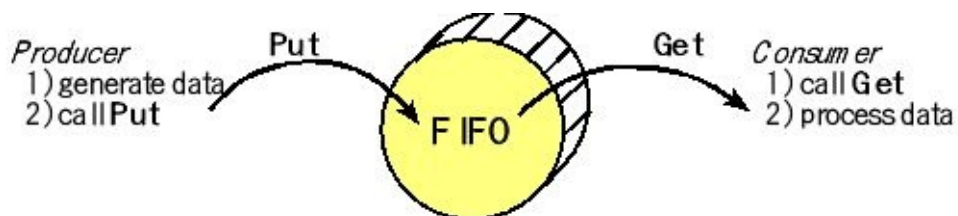
## Thread 2

## 4.3. First In First Out Queue

We introduced **first in first out circular queues (FIFO)** back in Chapter 2 when we presented interrupts. However, in this section we will delve deeper and investigate how operating systems use this important data structure. A common scenario in embedded systems has producers that generate data and consumers that process data. To decouple the producers and consumers from having to work in lock-step, a buffer is used to store the data, so a producer thread can produce when it can. All is fine as long as there is room in the buffer to store the produced data. Similarly, the consumer thread can process data when it can. Similarly, all is fine as long as the buffer is non-empty. A common implementation of such a buffer is the FIFO queue, which preserves the order of data, so that the first piece of data generated is the first consumed.

### 4.3.1. Producer/Consumer problem using a FIFO

The **FIFO** is quite useful for implementing a buffered I/O interface (Figure 4.5). The function **Put** will store data in the FIFO, and the function **Get** will remove data. It operates in a first in first out manner, meaning the **Get** function will return/remove the oldest data. It can be used for both buffered input and buffered output. This order-preserving data structure temporarily saves data created by the source (producer) before it is processed by the sink (consumer). The class of FIFOs studied in this section will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be carefully shared by more than one program. The advantage of using a FIFO structure for a data flow problem is that we can decouple the producer and consumer threads. Without the FIFO we would have to produce one piece of data, then process it, produce another piece of data, then process it. With the FIFO, the producer thread can continue to produce data without having to wait for the consumer to finish processing the previous data. This decoupling can significantly improve performance.



*Figure 4.5. The FIFO is used to buffer data between the producer and consumer. The number of data stored in the FIFO varies dynamically, where Put adds one data element and Get removes/returns one data element.*

Another name for the FIFO is bounded buffer. For example, a FIFO is used while

streaming audio from the Internet. As sound data are received from the Internet they are stored (calls **Put**) into a FIFO. When the sound board needs data it calls **Get**. As long as the FIFO never becomes full or empty, the sound is played in a continuous manner. A FIFO is also used when you ask the computer to print a file. Rather than waiting for the actual printing to occur character by character, the print command will put the data in a FIFO. Whenever the printer is free, it will get data from the FIFO. The advantage of the FIFO is it allows you to continue to use your computer while the printing occurs in the background. To implement this magic, our RTOS must be able to manage FIFOs. There are many producer/consumer applications, as we previously listed in Table 3.1, where the processes on the left are producers that create or input data, while the processes on the right are consumers which process or output data.

### 4.3.2. Little's Theorem

In this section we introduce some general theory about queues. Let  $N$  be the average number of data packets in the queue plus the one data packet currently being processed by the consumer. Basically,  $N$  is the average number of packets in the system. Let  $L$  be the average arrival rate in packets per second (pps). Let  $R$  be the average response time of a packet, which includes the time waiting in the queue plus the time for the consumer to process the packet. **Little's Theorem** states

$$N = L * R$$

As long as the system is stable, this result is not influenced by the probability distribution of the producer, the probability distribution of the consumer or the service order. Let  $S$  be the mean service time for a packet. Thus,  $C=1/S$  is defined as the **system capacity** (pps). Stable in this context means the packet arrival rate is less than the system capacity ( $L < C$ ). This means, in most cases, the queue length can be chosen so the queue never fills and no data are lost. In this case, the arrival rate  $L$  is also the output rate  $T$ , or throughput of the system. We can use Little's Theorem to estimate average response time,

$$R = N/T$$

In general, we want  $T$  to be high and  $R$  to be low. To handle these two conflicting goals, we develop the concept of a power metric for the queue. We can define **utilization factor** as the throughput divided by the capacity, which is a normalized throughput,

$$U = T/C$$

$U$  defines the loading of the queue, because it is. We can define **normalized mean response time**,  $R/S$ . We next define **power metric**  $P$  as utilization factor divided by normalized mean response time,

$$P = U/(R/S) = (T*S)/(R/S)$$

Substituting Little's Theorem ( $R=N/T$ ), we can write

$$P = U^2/N$$

The goal of the operating system is to maximize  $P$ .

### 4.3.3. FIFO implementation

FIFOs can be statically allocated, where the buffer size is fixed at compile time, Figure 4.6. This means the maximum number of elements that can be stored in the FIFO at any one time is determined at design time. Alternately, FIFOs can be dynamically allocated, where the OS allows the buffer to grow and shrink in size dynamically. To allow a buffer to grow and shrink, the system needs a memory manager or heap. A **heap** allows the system to allocate, deallocate, and reallocate buffers in RAM dynamically. There are many memory managers (heaps), but the usual one available in C has these three functions. The function **malloc** creates a new buffer of a given size. The function **free** deallocates a buffer that is no longer needed. The function **realloc** allocates a new buffer, copies data from a previous buffer into the new buffer of different size, and then deallocates the previous buffer. **realloc** is the function needed to increase or decrease the allocated space for the FIFO statically-allocated FIFOs might result in lost data or reduced bandwidth compared to dynamic allocation.

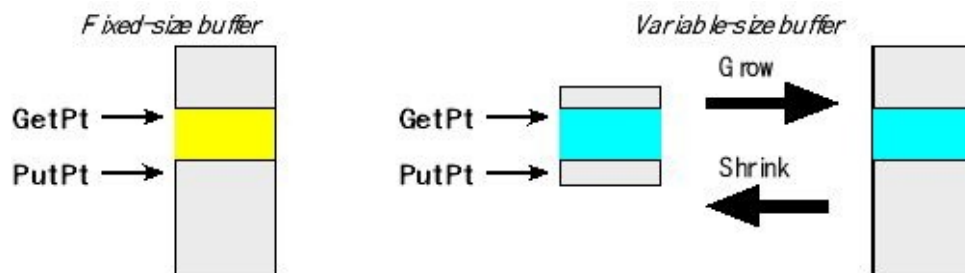


Figure 4.6. With static allocation, the maximum number of elements stored in the FIFO is fixed at compile time. With dynamic allocation, the system can call **realloc** when the FIFO is almost full to grow the size of the FIFO dynamically. Similarly, if the FIFO is almost empty, it can shrink the size freeing up memory.

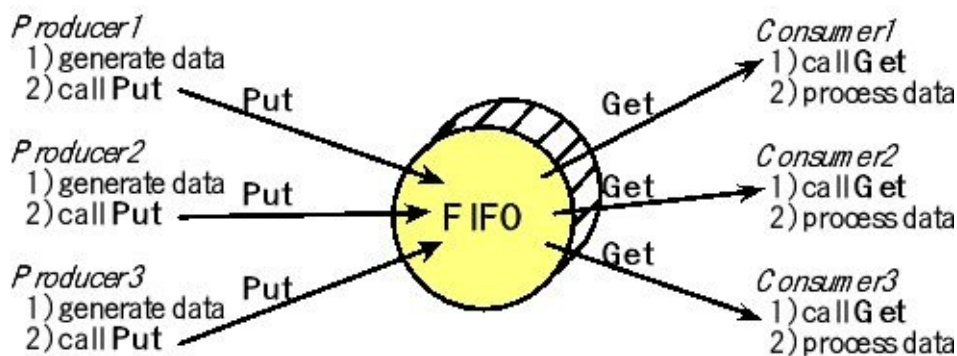
A system is considered to be **deterministic** if when the system is run with the same set of inputs, it produces identical responses. Most real-time systems often require deterministic behavior, because testing can be used to certify performance. Dynamically-allocated FIFOs cause the behavior of one subsystem (that might allocate large amounts of RAM from the heap) to affect behavior in another unrelated subsystem (our FIFO that wishes to increase buffer size). It is better for real-time systems to be reliable and verifiable than to have higher performance. As the heap

runs, it can become fragmented; meaning the free memory in the heap has many little pieces, rather than a few big pieces. Since the time to reallocate a buffer can vary tremendously, depending on the fragmentation of the heap, it will be difficult to predict execution time for the FIFO functions. Since a statically allocated FIFO is simple, we will be able to predict execution behavior. For these reasons, we will restrict FIFO construction to static allocation. In other words, you should not use **malloc** and **free** in your RTOS.

There are many ways to implement a statically-allocated FIFO. We can use either two pointers or two indices to access the data in the FIFO. We can either use or not use a counter that specifies how many entries are currently stored in the FIFO. There are even hardware implementations. For non-OS implementations of the FIFO, see Section 2.3. In this section we will present three implementations using semaphores.

#### 4.3.4. Three-semaphore FIFO implementation

The first scenario we will solve is where there are multiple producers and multiple consumers. In this case all threads are main threads, which are scheduled by the OS. The FIFO is used to pass data from the producers to the consumers. In this situation, the producers do not care to which consumer their data are passed, and the consumers do not care from which producer the data arrived. These are main threads, so we will block producers when the FIFO is full and we will block consumers when the FIFO is empty.



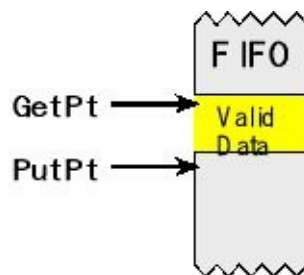
*Figure 4.7. FIFO used to pass data from multiple producers to multiple consumers. All threads are main threads.*

The producer puts data into the FIFO. If the FIFO is full and the user calls **Fifo\_Put**, there are two responses we could employ. The first response would be for the **Fifo\_Put** routine to block assuming it is unacceptable to discard data. The second response would be for the **Fifo\_Put** routine to discard the data and return with an error value. In this subsection we will block the producer on a full FIFO. This implementation can be used if the producer is a main thread, but cannot be used if the producer is an event thread or ISR.

The consumer removes data from the FIFO. For most applications, the consumer will

be a main thread that calls **Fifo\_Get** when it needs data to process. After a get, the particular information returned from the get routine is no longer saved in the FIFO. If the FIFO is empty and the user tries to get, the **Fifo\_Get** routine will block because we assume the consumer needs data to proceed. The FIFO is order preserving, such that the information returned by repeated calls to **Fifo\_Get** give data in the same order as the data saved by repeated calls of **Fifo\_Put** .

The two-pointer implementation has, of course, two pointers. If we were to have infinite memory, a FIFO implementation is easy (Figure 4.8). **GetPt** points to the data that will be removed by the next call to **Fifo\_Get** , and **PutPt** points to the empty space where the data will be stored by the next call to **Fifo\_Put** , see Program 4.4.



*Figure 4.8. The FIFO implementation with infinite memory.*

```

uint32_t volatile *PutPt; // put next
uint32_t volatile *GetPt; // get next
void Fifo_Put(uint32_t data){ // call by value
    *PutPt = data; // Put
    PutPt++; // next
}
uint32_t Fifo_Get(void){ uint32_t data;
    data = *GetPt; // return by reference
    GetPt++; // next
    return data; // true if success
}

```

*Program 4.4. Code fragments showing the basic idea of a FIFO.*

There are four modifications that are required to the above functions. If the FIFO is full when **Fifo\_Put** is called, then the function should block. Similarly, if the FIFO is empty when **Fifo\_Get** is called, then the function should block. **PutPt** must be wrapped back up to the top when it reaches the bottom (Figure 4.9).



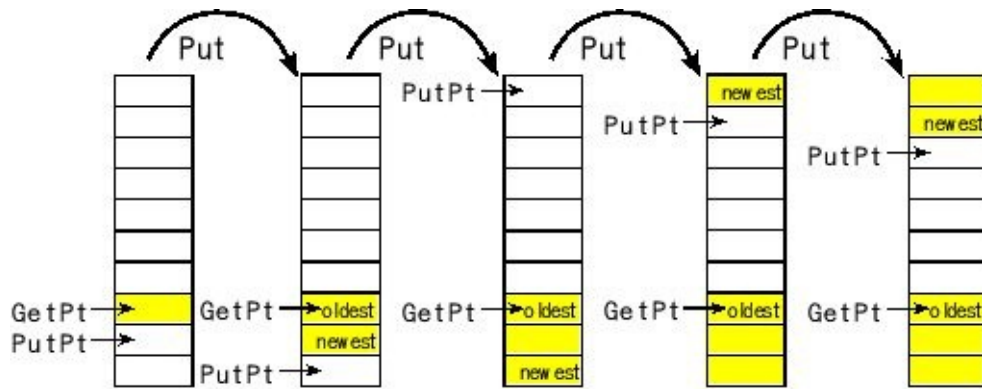


Figure 4.9. The FIFO *Fifo\_Put* operation showing the pointer wrap.

The **GetPt** must also be wrapped back up to the top when it reaches the bottom (Figure 4.10).

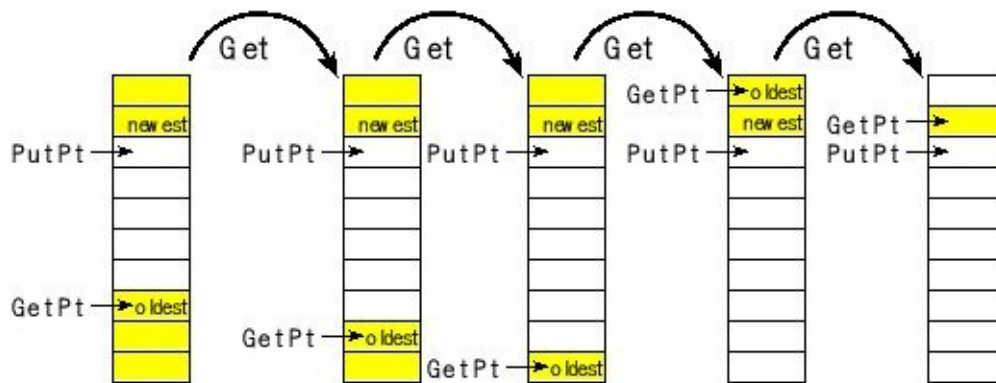


Figure 4.10. The FIFO *Fifo\_Get* operation showing the pointer wrap.

We will deploy two semaphores to describe the status of the FIFO, see Program 4.5. In this FIFO, each element is a 32-bit integer. The maximum number of elements, **FIFOSIZE**, is determined at compile time. In other words, to increase the allocation, we first change **FIFOSIZE**, and then recompile.

The first semaphore, **CurrentSize**, specifies the number of elements currently in the FIFO. This semaphore is initialized to zero, meaning the FIFO is initially empty, it is incremented by **Fifo\_Put** signifying one more element, and decremented by **Fifo\_Get** signifying one less element.

The second semaphore, **RoomLeft**, specifies the how many more elements could be put into the FIFO. This semaphore is initialized to **FIFOSIZE**, it is decremented by **Fifo\_Put** signifying there is space for one less element, and incremented by **Fifo\_Get** signifying there is space for one more element. When **RoomLeft** is zero, the FIFO is full.

Race conditions and critical sections are important issues in systems using interrupts. If there are more than one producer or more than one consumer, access to the pointers represent a critical section, and hence we will need to protect the pointers using a **FIFOmutex** semaphore.

```

#define FIFOSIZE 10    // can be any size
uint32_t volatile *PutPt; // put next
uint32_t volatile *GetPt; // get next
uint32_t static Fifo[FIFOSIZE];
int32_t CurrentSize;    // 0 means FIFO empty
int32_t RoomLeft;      // 0 means FIFO full
int32_t FIFOMutex;    // exclusive access to FIFO
// initialize FIFO
void OS_Fifo_Init(void){
    PutPt = GetPt = &Fifo[0]; // Empty
    OS_InitSemaphore(&CurrentSize, 0);
    OS_InitSemaphore(&RoomLeft, FIFOSIZE);
    OS_InitSemaphore(&FIFOMutex, 1);
}
void OS_Fifo_Put(uint32_t data){
    OS_Wait(&RoomLeft);
    OS_Wait(&FIFOMutex);
    *(PutPt) = data; // Put
    PutPt++;        // place to put next
    if(PutPt == &Fifo[FIFOSIZE]){
        PutPt = &Fifo[0]; // wrap
    }
    OS_Signal(&FIFOMutex);
    OS_Signal(&CurrentSize);
}
uint32_t OS_Fifo_Get(void){ uint32_t data;
    OS_Wait(&CurrentSize);
    OS_Wait(&FIFOMutex);
    data = *(GetPt); // get data
    GetPt++;        // points to next data to get
    if(GetPt == &Fifo[FIFOSIZE]){
        GetPt = &Fifo[0]; // wrap
    }
    OS_Signal(&FIFOMutex);
    OS_Signal(&RoomLeft);
    return data;
}

```

*Program 4.5. Two-pointer three-semaphore implementation of a FIFO. This implementation is appropriate when producers and consumers are main threads.*

**Checkpoint 4.4:** On average over the long term, what is the relationship between the number of times **Wait** is called compared to the number of times **Signal** is called?

**Checkpoint 4.5:** On average over the long term, what is the relationship between the number of times **Put** is successfully called compared to the number of times **Get** is successfully called? To answer this question, consider a successful call to **Put** as a call that correctly stored data, and a successful call to **Get** as a call that correctly returned data.

### 4.3.5. Two-semaphore FIFO implementation

If there is one producer as an event thread coupled with one or more consumers as main threads (Figure 4.11), the FIFO implementation shown in the previous section must be changed, because we cannot block or spin an event thread. If the FIFO is full when the producer calls **Put**, then that data will be lost. The number of times we lose data is recorded in **LostData**. The **Put** function returns an error (-1) if the data was not saved because the FIFO was full. This **Put** function cannot be called by multiple producers because of the read-modify-write sequence to **PutPt**. See Program 4.6. To tell if the FIFO is full, we simply compare the **CurrentSize** with its maximum. This is a statically allocated FIFO, so the maximum size is a constant.

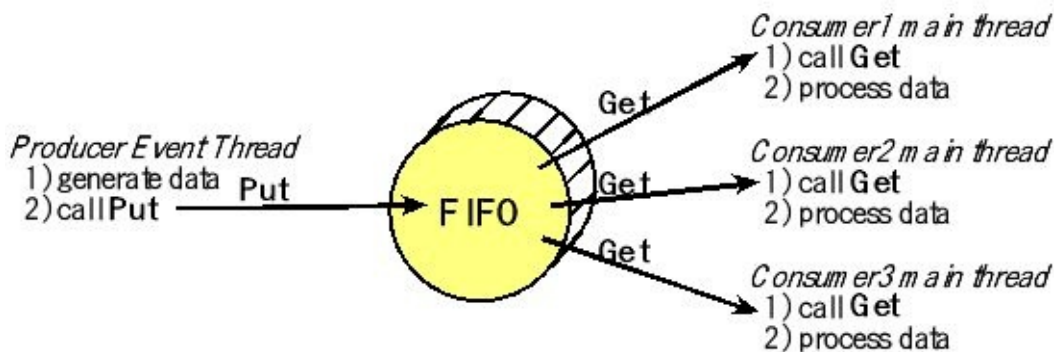


Figure 4.11. FIFO used to pass data from a single producer to multiple consumers. The producer is an event thread and the consumers are main threads.

```
#define FIFOSIZE 10    // can be any size
uint32_t volatile *PutPt; // put next
uint32_t volatile *GetPt; // get next
uint32_t static Fifo[FIFOSIZE];
int32_t CurrentSize;    // 0 means FIFO empty
int32_t FIFOmutex;     // exclusive access to FIFO
uint32_t LostData;
// initialize FIFO
void OS_Fifo_Init(void){
    PutPt = GetPt = &Fifo[0]; // Empty
    OS_InitSemaphore(&CurrentSize, 0);
    OS_InitSemaphore(&FIFOmutex, 1);
    LostData=0;
```

```

}
int OS_FIFO_Put(uint32_t data){
    if(CurrentSize == FIFOSIZE){
        LostData++;    // error
        return -1;
    }
    *(PutPt) = data;    // Put
    PutPt++;           // place for next
    if(PutPt == &Fifo[FIFOSIZE]){
        PutPt = &Fifo[0]; // wrap
    }
    OS_Signal(&CurrentSize);
    return 0;
}
uint32_t OS_FIFO_Get(void){uint32_t data;
    OS_Wait(&CurrentSize); // block if empty
    OS_Wait(&FIFOmutex);
    data = *(GetPt);    // get data
    GetPt++;           // points to next data to get
    if(GetPt == &Fifo[FIFOSIZE]){
        GetPt = &Fifo[0]; // wrap
    }
    OS_Signal(&FIFOmutex);
    return data;
}

```

*Program 4.6. Two-pointer two-semaphore implementation of a FIFO. This implementation is appropriate when a single producer is running as an event thread and multiple consumers are running as main threads.*

Note that, in this solution we no longer need the **RoomLeft** semaphore, which was used to protect the multiple changes to **PutPt** that multiple producers would entail. A single producer does not have this problem. We still need the **CurrentSize** semaphore because we have multiple consumers that can change the **GetPt** pointer. The **FIFOmutex** semaphore is needed to prevent two consumers from reading the same data.

### 4.3.6. One-semaphore FIFO implementation

If there is one producer as an event thread coupled with one consumer as a main thread (Figure 4.12), we can remove the mutex semaphore. This **Get** function cannot be called by multiple consumers because of the read-modify-write sequence to **GetI**. In the previous FIFO implementations, we used pointers, but in this example we use indices, see Program 4.7. Whether you use pointers versus indices is

a matter of style, and our advice is to use the mechanism you understand the best. As long as there is one event thread calling **Put** and one main thread calling **Get**, this implementation does not have any critical sections.

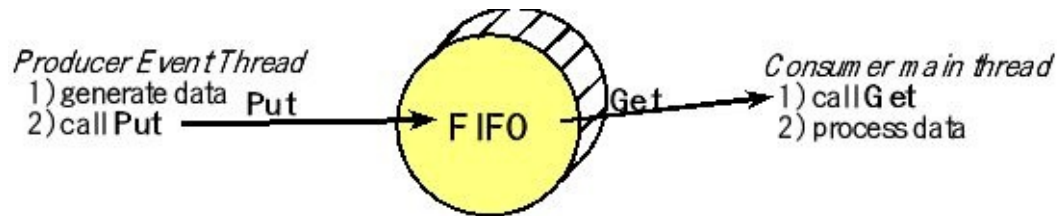


Figure 4.12. FIFO used to pass data from a single producer to a single consumer. The producer is an event thread and the consumer is a main thread.

```
#define FIFOSIZE 10 // can be any size
uint32_t PutI;    // index of where to put next
uint32_t GetI;   // index of where to get next
uint32_t Fifo[FIFOSIZE];
int32_t CurrentSize; // 0 means FIFO empty, FIFOSIZE means full
uint32_t LostData; // number of lost pieces of data
void OS_FIFO_Init(void){
    PutI = GetI = 0; // Empty
    OS_InitSemaphore(&CurrentSize, 0);
    LostData = 0;
}
int OS_FIFO_Put(uint32_t data){
    if(CurrentSize == FIFOSIZE){
        LostData++;
        return -1; // full
    } else{
        Fifo[PutI] = data;    // Put
        PutI = (PutI+1)%FIFOSIZE;
        OS_Signal(&CurrentSize);
        return 0; // success
    }
}
uint32_t OS_FIFO_Get(void){uint32_t data;
    OS_Wait(&CurrentSize); // block if empty
    data = Fifo[GetI];    // get
    GetI = (GetI+1)%FIFOSIZE; // place to get next
    return data;
}
```

Program 4.7. Two-index one-semaphore implementation of a FIFO. This implementation is appropriate when a single producer is running as an event thread and a single consumer is running as a main thread.

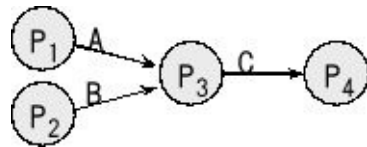
The use of indexes rather than pointers also means all index arithmetic is a simple modulo the size of the FIFO to implement the wraparound.

**Checkpoint 4.6:** Notice in Program 4.7 that there are two conditions that result in PutI equaling GetI. One condition is the FIFO is empty and the other condition is the FIFO is full. How does the software distinguish between these two conditions?

**Checkpoint 4.7:** How might you optimize Program 4.7 if the size of the FIFO were a power of 2?

### 4.3.7. Kahn Process Networks

Gilles Kahn first introduced the **Kahn Process Network** (KPN). We use KPNs to model distributed systems as well as signal processing systems. Each node represents a computation block communicating with other nodes through unbounded FIFO channels. The circles in Figure 4.13 are computational blocks and the arrows are FIFO queues. The resulting process network exhibits deterministic behavior that does not depend on the various computation or communication delays. As such, KPNs have found many applications in modeling embedded systems, high-performance computing systems, and computational tasks.



*Figure 4.13. A Kahn Process Network consists of process nodes linked by unbounded FIFO queues.*

For each FIFO, only one process puts, and only one process gets. Figure 4.13 shows a KPN with four processes and three edges (communication channels). Processes  $P_1$  and  $P_2$  are producers, generating data into channels A and B respectively. Process  $P_3$  consumes one token from channel A and another from channel B (in either order) and then produces one token into channel C. Process  $P_4$  is a consumer because it consumes tokens.

We can use a KPN to describe signal processing systems where infinite streams of data are transformed by processes executing in sequence or parallel. Streaming data means we input/analyze/output one data packet at a time without the desire to see the entire collection of data all at once. Despite parallel processes, multitasking or parallelism are not required for executing this model. In a KPN, processes communicate via unbounded FIFO channels. Processes read and write atomic data elements, or alternatively called **tokens**, from and to channels. The read token is equivalent to a FIFO get and the write token is a FIFO put. In a KPN, writing to a channel is **non-blocking**. This means we expect the put FIFO command to always succeed. In other words, the FIFO never becomes full. From a practical perspective,

we can use KPN modeling for situations where the FIFOs never actually do become full. Furthermore, the approximate behavior of a system can be still be deemed for systems where FIFO full errors are infrequent. For these approximations we could discard data with the FIFO becomes full on a put instead of waiting for there to be free space in the FIFO.

On the other hand, reading from a channel requires blocking. A process that reads from an empty channel will stall and can only continue when the channel contains sufficient data items (tokens). Processes are not allowed to test an input channel for existence of tokens without consuming them. Given a specific input (token) history for a process, the process must be deterministic so that it always produces the same outputs (tokens). Timing or execution order of processes must not affect the result and therefore testing input channels for tokens is forbidden.

In order to optimize execution some KPNs do allow testing input channels for emptiness as long as it does not affect outputs. It can be beneficial and/or possible to do something in advance rather than wait for a channel. In the example shown in Figure 4.13, process P3 must get from both channel A and channel B. The left side of Program 4.8 shows the process stalls if the AFifo is empty (even if there is data in the BFifo). If the first FIFO is empty, it might be efficient to see if there is data in the other FIFO to save time (right side of Program 4.8).

<pre> void Process3(void){ int32_t inA, inB, out; while(1){ while(AFifo_Get(&amp;inA));}; while(BFifo_Get(&amp;inB));}; out = compute(inA,inB); CFifo_Put(out); } } </pre>	<pre> void Process3(void){ int32_t inA, inB, out; while(1){ if(AFifo_Size()==0){ while(BFifo_Get(&amp;inB));}; while(AFifo_Get(&amp;inA));}; } else{ while(AFifo_Get(&amp;inA));}; while(BFifo_Get(&amp;inB));}; } out = compute(inA,inB); CFifo_Put(out); } } </pre>
--	---

*Program 4.8. Two C implementations of a process on a KPN. The one on the right is optimized.*

Processes of a KPN are **deterministic**. For the same input history, they must always produce exactly the same output. Processes can be modeled as sequential programs that do reads and writes to ports in any order or quantity as long as the determinism property is preserved.

KPN processes are **monotonic**, which means that they only need partial information of the input stream in order to produce partial information of the output stream.

Monotonicity allows parallelism. In a KPN there is a total order of events inside a signal. However, there is no order relation between events in different signals. Thus, KPNs are only partially ordered, which classifies them as an untimed model.



---

## 4.4. Thread sleeping

Sometimes a thread needs to wait for a fixed amount of time. We will implement an **OS\_Sleep** function that will make a thread dormant for a finite time. A thread in the **sleep state** will not be run. After the prescribed amount of time, the OS will make the thread active again. Sleeping would be used for tasks which are not real-time. In Program 4.9, the **PeriodicStuff** is run approximately once a second.

```
void Task(void){
    InitializationStuff();
    while(1){
        PeriodicStuff();
        OS_Sleep(ONE_SECOND); // go to sleep for 1 second
    }
}
```

*Program 4.9. This thread uses sleep to execute its task approximately once a second.*

To implement the sleep function, we could add a counter to each TCB and call it **Sleep**. If **Sleep** is zero, the thread is not sleeping and can be run, meaning it is either in the run or active state. If **Sleep** is nonzero, the thread is sleeping. We need to change the scheduler so that **RunPt** is updated with the next thread to run that is not sleeping and not blocked, see Program 4.10.

```
void Scheduler(void){
    RunPt = RunPt->next; // skip at least one
    while((RunPt->Sleep)||((RunPt-> blocked)){
        RunPt = RunPt->next; // find one not sleeping and not blocked
    }
}
```

*Program 4.10. Round-robin scheduler that skips threads if they are sleeping or blocked.*

Any thread with a nonzero **Sleep** counter will not be run. The user must be careful not to let all the threads go to sleep, because doing so would crash this implementation. Next, we need to add a periodic task that decrements the **Sleep** counter for any nonzero counter. When a thread wishes to sleep, it will set its **Sleep** counter and invoke the cooperative scheduler. The period of this decrementing task will determine the resolution of the parameter **time**.

Notice that this implementation is not an exact time delay. When the sleep parameter is decremented to 0, the thread is not immediately run. Rather, when the parameter reaches 0, the thread is signified ready to run. If there are  $n$  other threads in the TCB

list and the thread switch time is  $\Delta t$ , then it may take an additional  $n*\Delta t$  time for the thread to be launched after it awakens from sleeping.

## 4.5. Deadlocks

One of the drawbacks of semaphores is a deadlock. With a **deadlock** there is a circle of threads blocked (or spinning) because they are waiting on each other. There are four necessary conditions for a deadlock to occur.

- Mutual exclusion
- Hold and wait
- No preemption of resources
- Circular waiting

**Mutual exclusion** means one thread will have exclusive access to a resource and other threads will have to wait if they wish access to the resource. **Hold and wait** means a thread is allowed to hold one resource while it waits for another. A deadlock could be resolved if the operating system could detect the deadlock is about to occur and preempt resources by killing threads and recovering the resources attached to that thread. So, we say a necessary condition for a deadlock to occur is that the OS **does not support preemption** of resources. The last and most obvious condition for a deadlock to occur is **circular waiting**. Program 4.11 shows three threads that share three resources SDC, LCD, and CAN. To use a resource, a thread first requests the resource by waiting on its semaphore, uses the resource, and then releases the resource by signaling its semaphore.

Thread A	Thread B	Thread C
wait(&bLCD); //1	wait(&bSDC); //2	wait(&bCAN); //3
wait(&bSDC); //4	wait(&bCAN); //5	wait(&bLCD); //6
use LCD and SDC	use CAN and SDC	use CAN and LCD
signal(&bSDC);	signal(&bCAN);	signal(&bLCD);
signal(&bLCD);	signal(&bSDC);	signal(&bCAN);

*Program 4.11. A deadlock will occur if the execution sequence follows 1-2-3.*

One way to visualize a deadlock is to draw a **resource allocation graph**. Another name for this graph is a **wait-for graph**. Threads are drawn as circles and resources (binary semaphores) are drawn as rectangles. In these examples the resources are single instance. For example, there is only one CAN, one LCD, and one SDC. This means the mutual exclusive access is controlled by three binary semaphores. There are two types of arrows in a resource allocation graph. The steps 1,2,3 in Program

4.11 all successfully return from a wait on a binary semaphore. We signify a thread possessing a resource using an **assignment arrow** from the resource to the thread. The steps 4,5,6 in Program 4.11 all execute a wait on a binary semaphore, but do not return because the resource is unavailable. We signify a thread waiting for a resource using a **request arrow** from the thread to the resource. Notice that a thread can have at most one request arrow, because once it is spinning or blocked on a semaphore it will not continue to execute. A closed path in a single-instance resource allocation graph is an indication that a deadlock has occurred. Figure 4.14 plots the resource allocation graph occurring if the example in Program 4.11 executes steps 1,2,3,4,5,6.

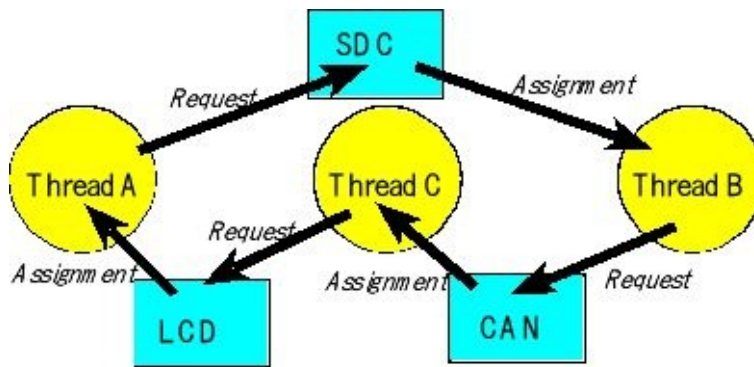


Figure 4.14. A resource allocation graph can be used to visualize a deadlock.

One way to **prevent deadlocks** is to remove one of the four necessary conditions required to have a deadlock. We could remove mutual exclusion by eliminating the semaphores all together. However, this usually impractical. One simple way to eliminate hold and wait is to request all resources at the same time. The OS will either grant a thread all its resources or block it until the resources are available. Notice in Program 4.12 that a new wait function is needed that supports multiple simultaneous requests. One disadvantage of this solution is the efficiency of requesting a resource before it may be needed.

Thread A	Thread B	Thread C
wait(&bLCD,&bSDC); use LCD and SDC	wait(&bSDC,&bCAN); use CAN and SDC	wait(&bCAN,&bLCD); use CAN and LCD
signal(&bLCD,&bSDC);	signal(&bSDC,&bCAN);	signal(&bCAN,&bLCD);

Program 4.12. A deadlock will not occur because there is no hold and wait.

Another way to prevent deadlocks is to remove the possibility of circular waiting. In this solution all resources are ordered numerically. A thread must request resources in this numerical order. If a thread needs resources 3, 6, and 15, the thread first asks for 3, then 6, and finally asks for 15. This solution like the first may cause a thread to request a resource before it is needed. In Program 4.13 we arbitrarily assign the LCD to 1, the SDC to 2, and the CAN to 3. In particular, we simply swap the order of

requesting in Thread 3 to conform to the numerical order and the possibility of deadlock is removed.

Thread A wait(&bLCD); wait(&bSDC); use LCD and SDC  signal(&bSDC);  signal(&bLCD);	Thread B wait(&bSDC); wait(&bCAN); use CAN and SDC  signal(&bCAN);  signal(&bSDC);	Thread C wait(&bLCD);  wait(&bCAN); use CAN and LCD  signal(&bLCD); signal(&bCAN);
--	--	--

*Program 4.13. A deadlock will not occur because there is no circular waiting.*

Deadlock prevention often puts severe restrictions on the operating system resulting in inefficiencies. A similar approach with far less restrictions is **deadlock avoidance**. With deadlock avoidance every time a thread requests a resource, it lists all the additional resources it might need to finish. If there is at least one safe sequence that allows all threads to complete, then the resource is granted. If no safe sequence can be found, the request would be denied. Referring back to Program 4.11 an operating system implementing deadlock avoidance would have denied Thread 3 at step 3 when it requests the CAN (knowing it also would need the LCD). It is a little inefficient to block thread 3 on the CAN even though the CAN was free. For more information about deadlock avoidance, search the term “Banker’s Algorithm”.

Another approach is to implement preemption. An operating system could use a resource allocation graph to detect that a deadlock has occurred. At this point, the OS would choose the least critical thread to kill that breaks the cycle. The resources would be recovered and the killed thread could be restarted. Another approach is to kill all the threads in the cycle and to restart them all.

A very effective approach to deadlock is to add timeouts to the wait function. For each wait, the thread specifies a maximum time it is willing to wait for a resource. If the timeout is triggered, the thread either skips that task or attempts to solve the task in another way.

---

## 4.6. Monitors

Semaphores are rich but low-level mechanism for synchronization. Semaphores are powerful, but when used incorrectly they can cause deadlocks and crashes. Monitors are a higher-level synchronization mechanism because proper use is enforced. Monitors can be developed to solve any of the applications presented in the previous section.

Semaphores are essentially shared global variables, which can be accessed anywhere in the software system by calling wait or signal. There is no formal connection between the semaphore and the data being controlled by the semaphore. Semaphores enforce no control or have any guarantee of proper usage. A **monitor** will encapsulate the data with synchronization mechanisms to access the data. A monitor defines a **lock** and zero or more condition variables for managing concurrent access to shared data. The monitor uses the lock to insure that only a single thread is active in the monitor at any instance. The lock also provides mutual exclusion for shared data. Condition variables enable threads to go to sleep inside of critical sections, by releasing their lock at the same time it puts the thread to sleep.

A monitor encapsulates protected data with synchronization. A thread acquires the mutex at the start by accessing the lock. Once granted acquiring the lock, the thread operates on the shared data. If the thread cannot complete, it will temporarily release the mutex, leaving the data in a consistent state. It will need to reacquire when it can continue. When complete the thread releases the mutex and exits the monitor.

A **condition variable** is a queue of threads waiting for something inside a critical section. Condition variables support three operations: **Wait** , **Signal** and **Broadcast** . Although monitors use functions called **Wait** and **Signal** , these are not the same operations as semaphores. The **Wait** function takes a lock parameter. In an atomic fashion it will either acquire the lock or go to sleep. When the process wakes up, it attempts to reacquire the lock. The **Signal** function wakes up a waiting thread, if one exists. Otherwise, it does nothing. The **Broadcast** function will wake up all waiting threads. A thread must hold the lock when doing condition variable operations.

To illustrate the concept, we will design a FIFO using a monitor for synchronization. This implementation handles the empty condition, but assumes the FIFO is never full. This FIFO has two public functions, **Put()** , which enters data into the FIFO, and **Get()** , which is used to extract data from the FIFO. There is a private lock, called Lock. Private means the lock cannot be accessed outside of the monitor. The FIFO of course also has private data, which is the queue. All access to the FIFO requires capturing the lock. If a thread finds the lock unavailable it will spin or block or sleep. It will attempt to reacquire the lock. After acquiring the lock, the **Put** operation will enter the item onto the queue, signal the conditionVar, and then release the lock. The **conditionVar->Signal** operation will wake up a thread

currently sleeping on the **conditionVar**. If there are no sleeping threads, then **Signal** has no action. Compare this to a semaphore **Signal**, which will increment its counter regardless of whether or not any consumer threads are waiting for data. The operation **Get** must all acquire the lock before proceeding. If the FIFO is empty, the thread will wait for data by releasing the lock and go to sleep. Notice that this thread is not holding any resources while it waits. When the sleeping thread awakens, it must attempt to reacquire the lock before proceeding to step 3 where data is removed from the queue.

<b>Put(item)</b> 1) lock->Acquire(); 2) put item on queue; 3) conditionVar->Signal(); 4) lock->Release();	<b>Get()</b> 1) lock->Acquire(); 2) while queue is empty conditionVar->Wait(lock); 3) remove item from queue; 4) lock->Release(); 5) return item;
---	---

Condition variables do not have any history, but semaphores do. When **Signal** is executed, if no thread is waiting, the signal is a no-op. If a thread executes **Wait**, it waits. The wait and signal functions of a semaphore are commutative; the result is the same regardless of the order of execution. In other words, if one thread calls a semaphore wait and another thread calls a semaphore signal, the result is the same regardless of the order of execution. Condition variables are not commutative. In other words, the order of execution matters. Therefore, all access to the Signal and Wait functions of a condition variable must require acquiring a lock.

There are two flavors of monitors that differ in the scheduling semantics of the function signal. With a **Hoare monitor** the signal function immediately switches from the thread that called signal to a waiting thread. The condition that the waiting thread was anticipating is guaranteed to hold when waiting thread executes. The thread that called Signal must make sure the data is in a consistent state before signaling. With a **Mesa monitor**, as implemented in Java, the function signal places a thread on the ready queue, but thread that called signal continues inside monitor. This means when the awakened thread eventually runs, the condition is not necessarily true. In a Mesa monitor after returning from a wait, the thread only knows something as changed.

<b>Hoare wait</b> if(FIFO empty) wait(condition)	<b>Mesa wait</b> while(FIFO empty) wait(condition)
--	--

---

## 4.7. Fixed Scheduling

In the round robin scheduler of the previous chapter, the threads were run one at a time and each was given the same time slice. When using semaphores the thread scheduler dynamically runs or blocks threads depending on conditions at that time. There is another application of thread scheduling sometimes found in real-time embedded systems, which involves a fixed scheduler. In this scheduler, the thread sequence and the allocated time-slices are determined *a priori*, during the design phase of the project. This class of problems is like creating the city bus schedule, managing a construction project, or routing packages through a warehouse. Because of this analogy, one would expect fundamental principles of managing a construction project will apply to the design of a fixed scheduler. It is important to have accurate data about the tasks in advance; we should build slack into the plan expecting delays and anticipating problems; “just in time” where tasks are performed when they are actually needed. First, we create a list of tasks to perform

1. Assigning a priority to each task,
2. Defining the resources required for each task,
3. Determining how often each task is to run, and
4. Estimating how long each task will require to complete.

Next, we compare resources required to run versus the available resources at each point. Since this chapter deals with time management, the only resource we will consider here is processor cycles. In more complex systems, we could consider other resources like memory, necessary data, and I/O channels. For real-time tasks we want to guarantee performance, so we must consider the worst case estimate of how long each task will take, so the schedule can be achieved 100% of the time. On the other hand, if it is acceptable to meet the scheduling requirement most of the time, we could consider the average time it takes to perform each task. Lastly, we schedule the run times for each tasks by assigning times for the highest priority tasks first, then shuffle the assignments like placing pieces in a puzzle until all real-time tasks are scheduled as required. The tasks that are not real-time can be scheduled in the remaining slots. If all real-time tasks cannot be scheduled, then a faster microcontroller will be required. The design of this type of fixed scheduler is illustrated with a design example, Figure 4.15.



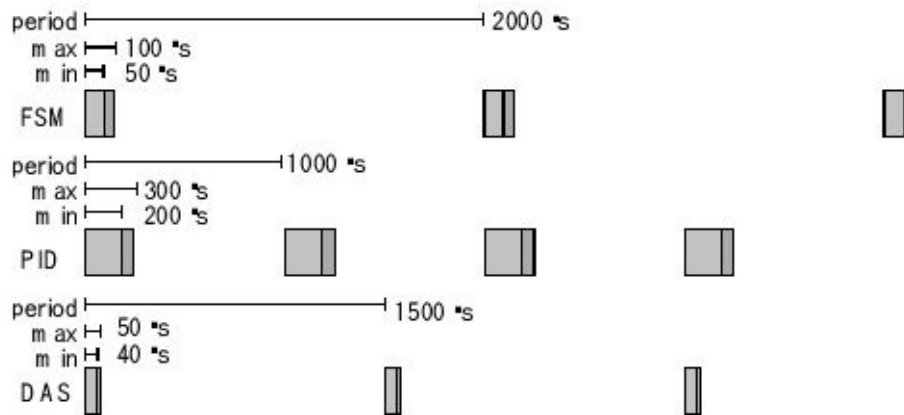


Figure 4.15. Real-time specifications for these three tasks.

The goal of this design example is to schedule three real-time tasks: a finite state machine ( **FSM** ), a proportional-integral-derivative controller ( **PID** ), and a data acquisition system ( **DAS** ). There will also be one non-real-time task, **PAN** , which will input/output with the front panel. Figure 4.15 shows that each real-time task in the example has a required period of execution, a maximum execution time, and a minimum execution time.

Because we wish to guarantee tasks will always be started on time, we will consider the maximum times. If a solution were to exist, then we will be able find one with a repeating 6000- $\mu$ s pattern, because 6000 is the least common multiple of 2000, 1000, and 1500. The basic approach to scheduling periodic tasks is to time-shift the second and third tasks so that when the three tasks are combined, there are no overlaps, as shown in Figure 4.16. We start with the most frequent task, which in this example is the PID controller, and then we schedule the FSM task immediately after it. In this example, about 41% of the time is allocated to real-time tasks. A solution is possible for this case because the number of tasks is small, and there is a simple 1/1.5/2 relationship between the required periods. Then, we schedule non real-time tasks in the remaining intervals.

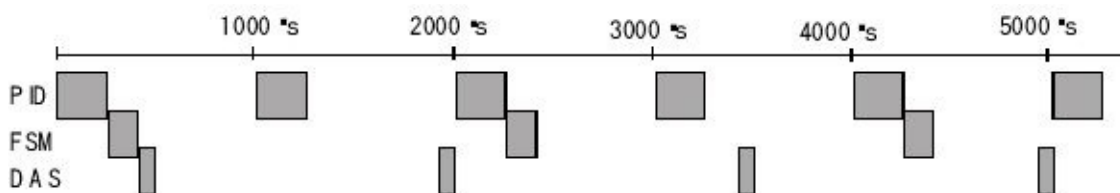


Figure 4.16. Repeating pattern to schedule these three real-time tasks.

Program 4.14 shows the four threads for this system. The real-time threads execute **OS\_Suspend** when it completes its task, which will suspend the thread and run the non-real-time thread. In this way, each thread will run one time through the **for** loop at the period requirement specified in Figure 4.15. When the threads explicitly release control (in this case by calling **OS\_Suspend** ), the system is called **cooperative multitasking**. The non-real-time thread ( **PAN** ) will be suspended by the timer interrupt, in a manner similar to the preemptive schedule described earlier in Section 4.1.

```

void FSM(void){ StatePtr Pt; uint8_t in;
Pt = SA;          // Initial State
for(;;) {
OS_Suspend();    // Runs every 2ms
Port_Out(Pt->Out); // Output depends on the current state
in = Port_In();
Pt = Pt->Next[in]; // Next state depends on the input
}
}
void PID(void){ uint8_t speed,power;
PID_Init();      // Initialize
for(;;) {
OS_Suspend();    // Runs every 1ms
speed = PID_In(); // read tachometer
power = PID_Calc(speed);
PID_Out(power);  // adjust power to motor
}
}
void DAS(void){ uint8_t raw;
DAS_Init();      // Initialize
for(;;) {
OS_Suspend();    // Runs every 1.5ms
raw = DAS_In();  // read ADC
Result = DAS_Calc(raw);
}
}
void PAN(void){ uint8_t input;
PAN_Init();      // Initialize
for(;;) {
input = PAN_In(); // front panel input
if(input){
PAN_Out(input);  // process
}
}
}

```

*Program 4.14. Four user threads (FixedScheduler\_xxx).*

Program 4.15 creates the four thread control blocks. In this system the TCBs are not linked together, but rather exist as a table of four entries, one for each thread. Each thread will have a total of 396bytes of stack, and the stack itself exists inside the TCB. The **RunPt** will point to the TCB of the currently running thread.

```

struct TCB{
uint32_t *StackPt; // Stack Pointer

```

```

uint32_t MoreStack[83]; // 396 bytes of stack
uint32_t InitialReg[14]; // R4-R11,R0-R3,R12,R14
uint32_t InitialPC; // pointer to program to execute
uint32_t InitialPSR; // 0x01000000
};
typedef struct TCB TCBType;
TCBType *RunPt; // thread currently running
#define TheFSM &sys[0] // finite state machine
#define ThePID &sys[1] // proportional-integral-derivative
#define TheDAS &sys[2] // data acquisition system
#define ThePAN &sys[3] // front panel
TCBType sys[4]={
  { &sys[0].InitialReg[0],{ 0}, (uint32_t) FSM, 0x01000000},
  { &sys[1].InitialReg[0],{ 0}, (uint32_t) PID, 0x01000000},
  { &sys[2].InitialReg[0],{ 0}, (uint32_t) DAS, 0x01000000},
  { &sys[3].InitialReg[0],{ 0}, (uint32_t) PAN, 0x01000000}
};

```

*Program 4.15. The thread control blocks (FixedScheduler\_xxx).*

Program 4.16 defines the data structure containing the details of the fixed scheduler. This structure is a circular linked list, because the schedule repeats. In particular, the 22 entries explicitly define the schedule drawn in Figure 4.16. The front panel thread ( PAN ) is assigned to run in the gaps when no real-time thread requires execution.

```

struct Node{
  struct Node *Next; // circular linked list
  TCBType *ThreadPt; // which thread to run
  uint32_t TimeSlice; // how long to run it
};
typedef struct Node NodeType;
NodeType *NodePt;
NodeType Schedule[22]={
  { &Schedule[1], ThePID, 300}, // interval 0, 300
  { &Schedule[2], TheFSM, 100}, // interval 300, 400
  { &Schedule[3], TheDAS, 50}, // interval 400, 450
  { &Schedule[4], ThePAN, 550}, // interval 450, 1000
  { &Schedule[5], ThePID, 300}, // interval 1000, 1300
  { &Schedule[6], ThePAN, 600}, // interval 1300, 1900
  { &Schedule[7], TheDAS, 50}, // interval 1900, 1950
  { &Schedule[8], ThePAN, 50}, // interval 1950, 2000
  { &Schedule[9], ThePID, 300}, // interval 2000, 2300
  { &Schedule[10],TheFSM, 100}, // interval 2300, 2400
  { &Schedule[11],ThePAN, 600}, // interval 2400, 3000

```

```

{ &Schedule[12],ThePID, 300}, // interval 3000, 3300
{ &Schedule[13],ThePAN, 100}, // interval 3300, 3400
{ &Schedule[14],TheDAS, 50}, // interval 3400, 3450
{ &Schedule[15],ThePAN, 550}, // interval 3450, 4000
{ &Schedule[16],ThePID, 300}, // interval 4000, 4300
{ &Schedule[17],TheFSM, 100}, // interval 4300, 4400
{ &Schedule[18],ThePAN, 500}, // interval 4400, 4900
{ &Schedule[19],TheDAS, 50}, // interval 4900, 4950
{ &Schedule[20],ThePAN, 50}, // interval 4950, 5000
{ &Schedule[21],ThePID, 300}, // interval 5000, 5300
{ &Schedule[0], ThePAN, 700} // interval 5300, 6000
};

```

*Program 4.16. The scheduler defines both the thread and the duration (FixedScheduler\_xxx.zip).*

A simple solution for the thread scheduler can be found on the book web site as **FixedScheduler\_xxx**. An **OS\_Suspend** creates the cooperative multitasking, and is used by the real-time threads when their task is complete. In this example, there is only one non-real-time thread, but it would be straight forward to implement a round-robin scheduler for these threads in the software interrupt handler.

We could have attempted to implement this system with regular periodic interrupts. In particular, we could have created three independent periodic interrupts and performed each task in a separate ISR. Unfortunately, there would be situations when one or more tasks would overlap. In other words, one interrupt might be requested while we are executing one of the other two ISRs. Although all tasks would run, some would be delayed. This delay is called **time-jitter**, which is defined as the difference between when a thread is supposed to run (see comments of Program 4.16) and when it does run. According to the Rate Monotonic Theorem we should have been able to schedule these tasks because

$$\sum_{i=0}^{n-1} \frac{E_i}{T_i} = \sum \frac{100}{2000} + \frac{300}{1000} + \frac{50}{1500} = 0.38 \leq n(2^{1/n} - 1) = 3(2^{1/3} - 1) = 0.78$$

---

## 4.8. Exercises

4.1 For each of the following terms give a definition in 16 words or less

a) active	g) deadlock	m) preemptive scheduler
b) atomic	h) hook	n) producer-consumer
c) blocked	i) nonreentrant	o) reentrant
d) bounded buffer	j) path expression	p) rendezvous
e) bounded waiting	k) sleeping	q) round robin scheduler
f) critical section	l) normalized mean response time	r) spin lock

4.2 Consider the queue of people waiting in line at the bank. How can Little's Theorem be used to measure the average time a person spends in the bank (time waiting plus time being served).

4.3 Consider situation of cars traveling across a bridge. Typically, 10 cars/sec arrive at the bridge. On a sunny day it takes 10 seconds to cross the bridge. Use Little's Theorem explain what happens on a rainy day when now it takes 100 seconds to cross the bridge.

4.4 Use Little's Theorem to explain why a fast food restaurant requires a smaller dining room than a regular restaurant even though they the same customer arrival rate.

4.5 If a thread is blocked because the output display is not available, when should you wake it up (signal it)?

4.6 You have three tasks. Task 1 takes a maximum of 1 ms to execute and runs every 10 ms. Task 2 takes a maximum of 0.5 ms to execute and runs every 1 ms. Task 3 takes a maximum of 1 ms to execute and runs every 100 ms. Do you think a scheduling algorithm exists? Justify your answer.

4.7 Consider a problem of running three foreground threads using a preemptive scheduler with semaphore synchronization. Each thread has a central **body()** containing code that should be executed together. The basic shell of this system is given. Define one or more semaphores, then add semaphore function calls to implement a *three-thread rendezvous*. Basically, each time through the **while** loop, the first two threads to finish their **start()** code will wait for the last thread to finish its **start()** code. Then, all three threads will be active at the same time as they execute their corresponding **body()**. You may call any if the semaphore functions defined in this book. You will allocate one or more semaphores and add calls to semaphore functions, otherwise no other changes are allowed. You may assume

**thread1** runs first. For each semaphore you add, explain what it means to be 0, 1 etc.

<pre>void thread1(void){ init1(); while(1){ start1(); body1(); end1(); } }</pre>	<pre>void thread2(void){ init2(); while(1){ start2(); body2(); end2(); } }</pre>	<pre>void thread3(void){ init3(); while(1){ start3(); body3(); end3(); } }</pre>
--	--	--

**4.8** Consider a problem of deadlocks that can occur with semaphore synchronization. The following is a classic example that might occur if two threads need both the disk and the printer. In this example, the disk has a binary semaphore **DiskFree**, which is 1 if the disk is available, and similarly the printer has a binary semaphore **PrinterFree**, which is 1 if the printer is available. A deadlock occurs if each thread gets one resource then waits (on each other) for the other resource. In this example, we assume there is one disk and one printer.

<pre>void thread1(void){ OS_bWait(&amp;DiskFree); OS_bWait(&amp;PrinterFree);  // use disk and printer  OS_bSignal(&amp;DiskFree); OS_bSignal(&amp;PrinterFree); }</pre>	<pre>void thread2(void){ OS_bWait(&amp;PrinterFree); OS_bWait(&amp;DiskFree);  // use printer and disk  OS_bSignal(&amp;PrinterFree); OS_bSignal(&amp;DiskFree); }</pre>
--	--

In this problem we will develop a graphical method (called a *resource allocation graph*) to visualize/recognize the deadlock. Draw each thread in your system as an oval, and each binary semaphore as a rectangle. If a thread calls **OS\_bWait** and returns, then draw an arrow (called an *allocation edge*) from the semaphore to the thread. An arrow from a semaphore to a thread means that thread owns the resource. If a thread calls **OS\_bSignal**, then erase the previously drawn allocation edge. If a thread calls **OS\_bWait** and spins or blocks because the semaphore is not free, then draw an arrow from the thread to the semaphore (called a *request edge*). An arrow from a thread to a semaphore means that thread is waiting for the resource associated with the semaphore.

- a) Draw the resource allocation graph that occurs with the deadlock sequence
  - 1) thread1 executes **OS\_bWait(&DiskFree);**
  - 2) thread2 executes **OS\_bWait(&PrinterFree);**
  - 3) thread2 executes **OS\_bWait(&DiskFree);**
  - 4) thread1 executes **OS\_bWait(&PrinterFree);**
- b) This method can be generalized to detect that a deadlock has occurred with an

arbitrary number of binary semaphores and threads. What shape in the resource allocation graph defines a deadlock? In other words, generalize the use of this method such that you can claim

*“There is a deadlock if and only if the resource allocation graph contains a shape in the form of a \_\_\_\_\_”.*

c) Justify your answer by giving a deadlock example with three threads and three binary semaphores. In particular, give 1) the C code; 2) the execution sequence; 3) the resource allocation graph

**4.9** You are given three identical I/O ports to manage on the LM3S/TM4C, PortF, PortG, and PortH. You may assume there is a preemptive thread scheduler and blocking semaphores.

a) Look up the address of each port and its direction register.

b) Create a data structure to hold an address of the port and the address of the data direction register. Assume the type of this structure is called **PortType** .

c) Design and implement a manager that supports two functions. The first function is called **NewPort** . Its prototype is

**PortType \*NewPort(void);**

If a port is available when a thread calls **NewPort** , then a pointer to the structure, defined in part b) is returned. If no port is available, then the thread will block. When a port becomes available this thread will be awakened and the pointer to the structure will be returned. You may define and use blocking semaphores without showing the implementation of the semaphore or scheduler. The second function is called **FreePort** , and its prototype is

**void FreePort(PortType \*pt);**

This function returns a port so that it can be used by the other threads. Include a function that initializes the system, where all five ports are free. Hint: the solution is very similar to the FIFO queue example shown in Section 4.3.

**4.10** Consider a system with two LCD message displays in the context of a preemptive thread scheduler with blocking semaphores. To display a message, the OS can call either **LCD1\_OutString** or **LCD2\_OutString** passing it an ASCII string. These routines have critical sections but must run with interrupts enabled. The foreground threads will not call **LCD1\_OutString** or **LCD2\_OutString** directly; rather, the threads call a generic OS routine **OS\_Display** . If an LCD is free, the OS passes the message to the free LCD. If both LCDs are busy, the thread will block. There are many threads that wish to display messages, and the threads do not care or know onto which LCD their message will be displayed. You are given the **LCD1\_OutString** or **LCD2\_OutString** routines, the OS and the blocking semaphores with the following prototypes.

```
void LCD1_OutString(char *string); // up to 20ms to complete  
void LCD2_OutString(char *string); // up to 20ms to complete  
int OS_InitSemaphore(Sema4Type *semaPt, int16_t value);  
void OS_Wait(Sema4Type *semaPt);
```

**void OS\_Signal(Sema4Type \*semaPt);**

**a)** List the semaphores and private global variables needed for your solution. For each semaphore define what it means and what initial value it should have. Give the meaning and initial values for any private global variables you need. The threads will not directly access these semaphores or variables.

**b)** Write the generic OS display routine that the foreground threads will call (you may not disable interrupts or call any other functions other than the five functions shown above)

**void OS\_Display(char \*string){**



# 5. Real-time Systems

## Chapter 5 objectives are to:

- Review real-time applications that require priority
- Implement a priority scheduler
- Use the operating system to debounce switches
- Run event threads as high priority main threads
- Review of other real-time operating systems

The key concept in this chapter is the introduction of “priority”, which captures the relative importance of tasks in a system. Real-time systems in general and operating systems for real-time systems in particular use priority as a means to achieve effective performance. First we motivate the need for priority and then we will show how priority can be incorporated into our simple RTOS. We will conclude by reviewing how priority is implemented in some of the RTOS schedulers in popular use.

# 5.1. Data Acquisition Systems

To motivate the need for priority we will discuss some classic real-time system scenarios like Data Acquisition systems, Digital Signal Processing (DSP), and Real-Time Control systems. The level of detail provided here is not needed for the course, but we believe it will give you a context for the kinds of systems you may encounter as a practitioner in the RTOS domain.

## 5.1.1. Approach

Figure 5.1 illustrates the integrated approach to data acquisition systems. In this section, we begin with the clear understanding of the problem. We can use the definitions in this section to clarify the design parameters as well as to report the performance specifications.

The **measurand** is the physical quantity, property, or condition that the instrument measures. See Figure 5.2. The measurand can be inherent to the object (like position, mass, or color), located on the surface of the object (like the human EKG, or surface temperature), located within the object (e.g., fluid pressure, or internal temperature), or separated from the object (like emitted radiation.)

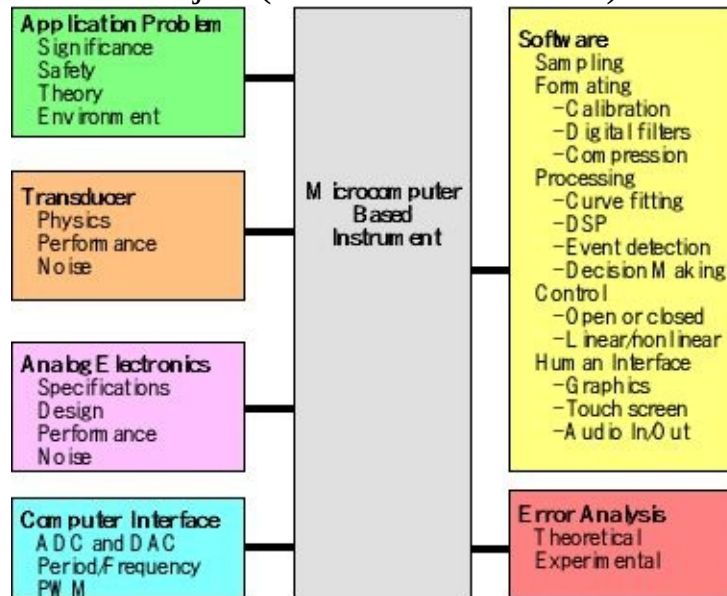
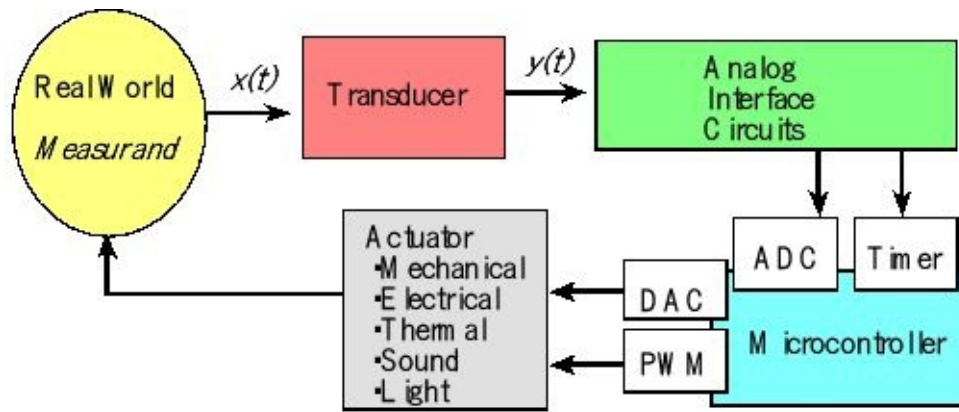


Figure 5.1. Individual components are integrated into a data acquisition system.



*Figure 5.2. Signal paths for a data acquisition system without an actuator; the control system includes an actuator so the system can use feedback to drive the real-world parameter to a desired state.*

In general, a **transducer** converts one energy type into another. In the context of this section, the transducer converts the measurand into an electrical signal that can be processed by the microcontroller-based instrument. Typically, a transducer has a primary sensing element and a variable conversion element. The primary sensing element interfaces directly to the object and converts the measurand into a more convenient energy form. The output of the variable conversion element is an electrical signal that depends on the measurand. For example, the primary sensing element of a pressure transducer is the diaphragm, which converts pressure into a displacement of a plunger. The variable conversion element is a strain gauge that converts the plunger displacement into a change in electrical resistance. If the strain gauge is placed in a bridge circuit, the voltage output is directly proportional to the pressure. Some transducers perform a direct conversion without having a separate primary sensing element and variable conversion element. The system contains **signal processing**, which manipulates the transducer signal output to select, enhance, or translate the signal to perform the desired function, usually in the presence of disturbing factors. The signal processing can be divided into stages. The **analog signal processing** consists of instrumentation electronics, isolation amplifiers, amplifiers, analog filters, and analog calculations. The first analog processing involves calibration signals and preamplification. Calibration is necessary to produce accurate results. An example of a calibration signal is the reference junction of a thermocouple. The second stage of the analog signal processing includes filtering and range conversion. The analog signal range should match the ADC analog input range. Examples of analog calculations include: RMS calculation, integration, differentiation, peak detection, threshold detection, phase lock loops, AM FM modulation/demodulation, and the arithmetic calculations of addition, subtraction, multiplication, division, and square root. When period, pulse width, or frequency measurement is used, we typically use an analog comparator to create a digital logic signal to measure. Whereas the Figure 5.1 outlined design components, Figure 5.2 shows the data flow graph for a data acquisition system or control system. The **control system** uses an actuator to drive a parameter in the real world to a desired

value while the data acquisition system has no actuator because it simply measures the parameter in a nonintrusive manner.

The **data conversion element** performs the conversion between the analog and digital domains. This part of the instrument includes: hardware and software computer interfaces, ADC, DAC, and calibration references. The analog to digital converter (**ADC**) converts the analog signal into a digital number. The digital to analog converter (**DAC**) converts a digital number to an analog output.

In many systems the input could be digital rather than analog. For these systems measuring period, pulse width, and/or frequency provides a low-cost high-precision alternative to the traditional ADC. Similarly, the output of the system could be digital. The **pulse width modulator (PWM)** is a digital output with a constant period, but variable duty cycle. The software can adjust the output of the actuator by setting the duty cycle of the PWM output.

The **digital signal processing** includes: data acquisition (sampling the signal at a fixed rate), data formatting (scaling, calibration), data processing (filtering, curve fitting, FFT, event detection, decision making, analysis), control algorithms (open or closed loop). The **human interface** includes the input and output which is available to the human operator. The advantage of computer-based instrumentation is that, devices that are sophisticated but easy to use and understand are possible. The **inputs** to the instrument can be audio (voice), visual (light pens, cameras), or tactile (keyboards, touch screens, buttons, switches, joysticks, roller balls). The **outputs** from the instrument can be numeric displays, CRT screens, graphs, buzzers, bells, lights, and voice.

## 5.1.2. Performance Metrics

Before designing a data acquisition system (DAS) we must have a clear understanding of the system goals. We can classify system as a **Quantitative DAS**, if the specifications can be defined explicitly in terms of desired range ( $r_x$ ), resolution ( $\Delta x$ ), precision ( $n_x$ ), and frequencies of interest ( $f_{\min}$  to  $f_{\max}$ ). If the specifications are more loosely defined, we classify it as a **Qualitative DAS**. Examples of qualitative systems include those which mimic the human senses where the specifications are defined using terms like “sounds good”, “looks pretty”, and “feels right.” Other qualitative systems involve the detection of events. We will consider two examples, a burglar detector, and an instrument to diagnose cancer. For binary detection systems like the presence/absence of a burglar or the presence/absence of cancer, we define a true positive (TP) when the condition exists (there is a burglar) and the system properly detects it (the alarm rings.) We define a false positive (FP) when the condition does not exist (there is no burglar) but the system thinks there is (the alarm rings.) A false negative (FN) occurs when the condition exists (there is a burglar) but the system does not think there is (the alarm is silent.) A true negative (TN) occurs when the condition does not exist (the patient does not have cancer) and

the system properly detects it (the instrument says the patient is normal.) **Prevalence** is the probability the condition exists, sometimes called pre-test probability. In the case of diagnosing the disease, prevalence tells us what percentage of the population has the disease. **Sensitivity** is the fraction of properly detected events (a burglar comes and the alarm rings) over the total number of events (number of robberies.) It is a measure of how well our system can detect an event. For the burglar detector, a sensitivity of 1 means when a burglar breaks in the alarm will go off. For the diagnostic instrument, a sensitivity of 1 means every sick patient will get treatment. **Specificity** is the fraction of properly handled non-events (a patient doesn't have cancer and the instrument claims the patient is normal) over the total number of non-events (the number of normal patients.) A specificity of 1 means no people will be treated for a cancer they don't have. The **positive predictive value** of a system (PPV) is the probability that the condition exists when restricted to those cases where the instrument says it exists. It is a measure of how much we believe the system is correct when it says it has detected an event. A PPV of 1 means when the alarm rings, the police will come and arrest a burglar. Similarly, a PPV of 1 means if our instrument says a patient has the disease, then that patient is sick. The **negative predictive value** of a system (NPV) is the probability that the condition does not exist when restricted to those cases where the instrument says it doesn't exist. A NPV of 1 means if our instrument says a patient doesn't have cancer, then that patient is not sick. Sometimes the true negative condition doesn't really exist (how many times a day does a burglar not show up at your house?) If there are no true negatives, only sensitivity and PPV are relevant.

$$\text{Prevalence} = (TP + FN) / (TP + TN + FP + FN)$$

$$\text{Sensitivity} = TP / (TP + FN)$$

$$\text{Specificity} = TN / (TN + FP)$$

$$\text{PPV} = TP / (TP + FP)$$

$$\text{NPV} = TN / (TN + FN)$$

There are two errors introduced by the sampling process. **Voltage quantizing** is caused by the finite word size of the ADC. The **precision** is determined by the number of bits in the ADC. If the ADC has  $n$  bits, then the number of distinguishable alternatives is

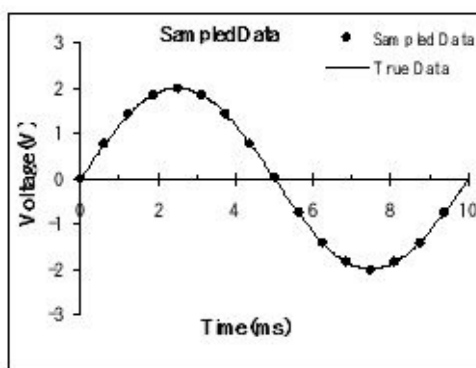
$$n_z = 2^n$$

**Time quantizing** is caused by the finite discrete sampling interval. The **Nyquist Theorem** states that if the signal is sampled at  $f_s$ , then the digital samples only contain frequency components from 0 to  $0.5 f_s$ . Conversely, if the analog signal does contain frequency components larger than  $\frac{1}{2} f_s$ , then there will be an **aliasing** error. Aliasing is when the digital signal appears to have a different frequency than the original analog signal. Simply put, if one samples a sine wave at a sampling rate of  $f_s$ ,

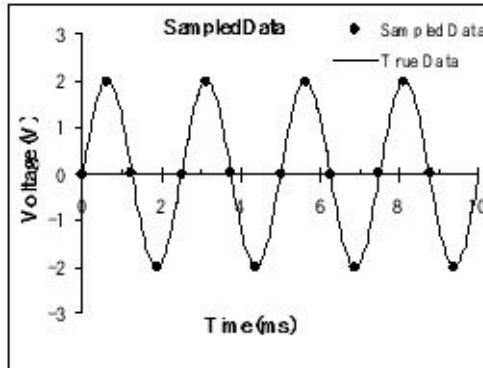
$$V(t) = A \sin(2\pi ft + \varphi)$$

is it possible to determine  $A$ ,  $f$  and  $\varphi$  from the digital samples? Nyquist Theory says that if  $f_s$  is strictly greater than twice  $f$ , then one can determine  $A$ ,  $f$  and  $\varphi$  from the digital samples. In other words, the entire analog signal can be reconstructed from the digital samples. But if  $f_s$  less than or equal to  $f$ , then one cannot determine  $A$ ,  $f$  and  $\varphi$ . In this case, the apparent frequency, as predicted by analyzing the digital samples, will be shifted to a frequency between 0 and  $\frac{1}{2} f_s$ .

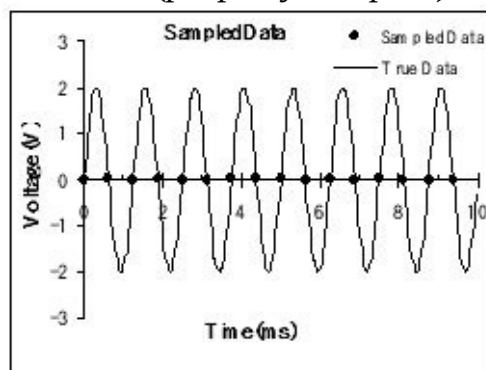
In Figure 5.3, the sampling rate is fixed at 1600 Hz and the signal frequency is varied. When sampling rate is exactly twice the input frequency, the original signal may or may not be properly reconstructed. In this specific case, it is frequency shifted (aliased) to DC and lost. When sampling rate is slower than twice the input frequency, the original signal cannot be properly reconstructed. It is frequency shifted (aliased) to a frequency between 0 and  $\frac{1}{2} f_s$ . In this case the 1500 Hz wave was aliased to 100 Hz.



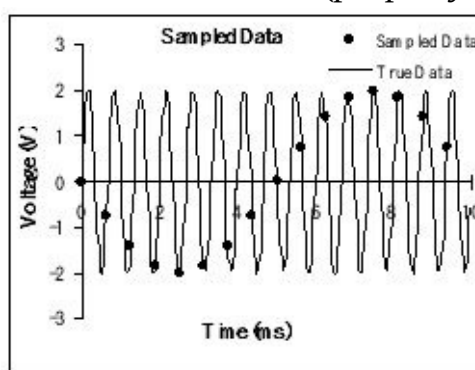
100 Hz sine wave (properly sampled)



400 Hz sine wave (properly sampled)



800 Hz sine wave (aliased)



1500 Hz sine wave (aliased)

Figure 5.3. Aliasing does not occur when the sampling rate is more than twice the signal frequency.

The choice of **sampling rate**,  $f_s$ , is determined by the maximum useful frequency contained in the signal. One must sample at least twice this maximum useful frequency. Faster sampling rates may be required to implement a digital filter and other digital signal processing.

$$f_s > 2 f_{max}$$

Even though the largest signal frequency of interest is  $f_{max}$ , there may be significant signal magnitudes at frequencies above  $f_{max}$ . These signals may arise from the input  $x$ , from added noise in the transducer or from added noise in the analog processing. Once the sampling rate is chosen at  $f_s$ , then a low pass analog filter may be required to remove frequency components above  $\frac{1}{2}f_s$ . A digital filter cannot be used to remove aliasing.

An interesting question arises: how do we determine the maximum frequency component in our input? If we know enough about our system, we might be able to derive an equation to determine the maximum frequency. For example, if a mechanical system consists of a mass, friction and a spring, then we can write a differential equation relating the applied force to the position of the object. The second way to find the maximum frequency component in our signal is to measure it with a spectrum analyzer.

**Valvano Postulate:** If  $f_{max}$  is the largest frequency component of the analog signal, then you must sample more than ten times  $f_{max}$  in order for the reconstructed digital samples to look like the original signal when plotted on a voltage versus time graph.

The choice of the **ADC precision** is a compromise of various factors. The desired resolution of the data acquisition system will dictate the number of ADC bits required. If the transducer is nonlinear, then the ADC precision must be larger than the precision specified in the problem statement. For example, let  $y$  be the transducer output, and let  $x$  be the real world signal. Assume for now, that the transducer output is connected to the ADC input. Let the range of  $x$  be  $r_x$ . Let the range of  $y$  be  $r_y$ . Let the required precision of  $x$  be  $n_x$ . The resolutions of  $x$  and  $y$  are  $\Delta_x$  and  $\Delta_y$  respectively. Let the following describe the nonlinear transducer.

$$y = f(x)$$

The required ADC precision,  $n_y$ , (in alternatives) can be calculated by:

$$\Delta_x = r_x / n_x$$

$$\Delta_y = \min \{f(x+\Delta_x) - f(x)\} \text{ for all } x \text{ in } r_x$$

$$n_y = r_y / \Delta_y$$

In general, we wish the analog signal processing to map the full scale range of the transducer into the full scale range of the ADC. If the ADC precision is  $N=2^n$  in alternatives, and the output impedance of the transducer is  $R_{out}$ , then we need an input impedance larger than  $N \cdot R_{out}$  to avoid loading the signal. We need the analog circuit to pass the frequencies of interest. When considering noise, we determine the signal

equivalent noise. For example, consider a system that measures temperature. If we wish to consider noise on signal  $V_{out}$ , we calculate the relationship between input temperature  $T$  and the signal  $V_{out}$ . Next, we determine the sensitivity of the signal to temperature,  $dV_{out}/dT$ . If the noise is  $V_n$ , then the temperature equivalent noise is  $T_n = V_n / (dV_{out}/dT)$ . In general, we wish all equivalent noises to be less than the system resolution.

An **analog low pass filter** may be required to remove aliasing. The cutoff of this analog filter should be less than  $\frac{1}{2}f_s$ . Some transducers automatically remove these unwanted frequency components. For example, a thermistor is inherently a low pass device. Other types of filters (analog and digital) may be used to solve the data acquisition system objective. One useful filter is a 60 Hz bandreject filter.

In order to prevent aliasing, one must know the frequency spectrum of the ADC input voltage. This information can be measured with a spectrum analyzer. Typically, a spectrum analyzer samples the analog signal at a very high rate ( $>1$  MHz), performs a Discrete Fourier Transform (DFT), and displays the signal magnitude versus frequency. We define  $z(t)$  as the input to the ADC. Let  $|Z(f)|$  be the magnitude of the ADC input voltage as a function of frequency. There are 3 regions in the magnitude versus frequency graph shown in Figure 5.4. We will classify any signal with amplitude less than the ADC resolution,  $\Delta_z$ , to be undetectable. This region is labeled “Undetectable”. Undetectable signals cannot cause aliasing regardless of their frequency. We will classify any signal with amplitude larger than the ADC resolution at frequencies less than  $\frac{1}{2}f_s$  to be properly sampled. This region is labeled “Properly sampled”. It is information in this region that is available to the software for digital processing. The last region includes signals with amplitude above the ADC resolution at frequencies greater than or equal to  $\frac{1}{2}f_s$ . Signals in this region will be aliased, which means their apparent frequencies will be shifted into the 0 to  $\frac{1}{2}f_s$  range.

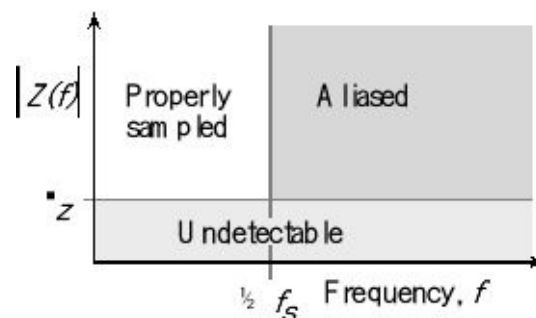


Figure 5.4. To prevent aliasing there should be no measurable signal above  $\frac{1}{2} f_s$ .

Most spectrum analyzers give the output in decibels full scale ( $dB_{FS}$ ). For an ADC system with a range of 0 to 3.3V, the full scale peak-to-peak amplitude for any AC signal is 3.3 V. If  $V$  is the DFT output magnitude in volts

$$dB_{FS} = 20 \log_{10}(V/3.3)$$



Table 5.1 calculates the ADC resolution in  $\text{dB}_{FS}$ . For a real ADC, the resolution will be a function of other factors other than bits. For example, the MAX1246 12-bit ADC has a minimum Signal-to-Noise+Distortion Ratio (SINAD) of 70 dB, meaning it is not quite 12 bits. The typical SINAD is 73 dB, which is slightly better than 12 bits.

<i>Bits</i>	$\text{dB}_{FS}$
8	-48.2
9	-54.2
10	-60.2
11	-66.2
12	-72.2
13	-78.3
14	-84.3

**Table 5.1. ADC resolution in  $\text{dB}_{FS}$ , assuming full scale is defined as peak-to-peak voltage.**

Aliasing will occur if  $|Z|$  is larger than the ADC resolution for any frequency larger than or equal to  $\frac{1}{2}f_s$ . In order to prevent aliasing,  $|Z|$  must be less than the ADC resolution. Our design constraint will include a safety factor of  $\alpha \leq 1$ . Thus, to prevent aliasing we will make:

$$|Z| < \alpha \Delta_z \quad \text{for all frequencies larger than or equal to } \frac{1}{2}f_s$$

This condition usually be can be satisfied by increasing the sampling rate or increasing the number of poles in the analog low pass filter. We cannot remove aliasing with a digital low pass filter, because once the high frequency signals are shifted into the 0 to  $\frac{1}{2}f_s$  range, we will be unable to separate the aliased signals from the regular ones. To determine  $\alpha$ , the sum of all errors (e.g., ADC, aliasing, and noise) must be less than the desired resolution.

To measure **resolution**, we use the **student's t-test** to determine if the system is able to detect the change. To use the student's t test we need to make the following assumptions:

- 1) Errors in one data set are independent, not correlated to errors in the other data set;
- 2) Errors in each data sample are independent, not correlated to errors within that set;
- 3) Errors are normally distributed;
- 4) Variance is unknown;
- 5) Variances in the two sets are equal.

We measure the input  $N$  times with the input fixed at  $x$  ( $X_{0i}$ ). Then, we measure it  $N$  more times with the input fixed at  $x+\Delta x$  ( $X_{1i}$ ). See Figure 5.5. We employ a test statistic to test the hypothesis  $H_0: \mu_0 = \mu_1$ . First, we estimate the means and variances of the data (assuming equal sized samples)

$$\bar{X}_0 = \frac{1}{N} \sum_i X_{0i}$$

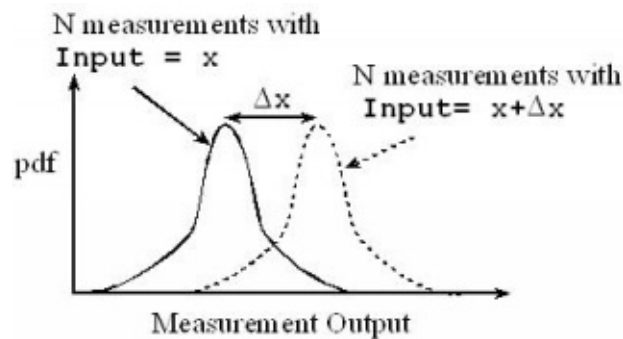
$$S_0^2 = \frac{1}{N-1} \sum_i (X_{0i} - \bar{X}_0)^2$$

$$\bar{X}_1 = \frac{1}{N} \sum_i X_{1i}$$

$$S_1^2 = \frac{1}{N-1} \sum_i (X_{1i} - \bar{X}_1)^2$$

From these, we calculate the test statistic  $t$ :

$$t = \frac{\bar{X}_1 - \bar{X}_0}{\sqrt{S_0^2/N + S_1^2/N}}$$



*Figure 5.5. Resolution means if the input increases by  $\Delta x$ , the system will probably notice.*

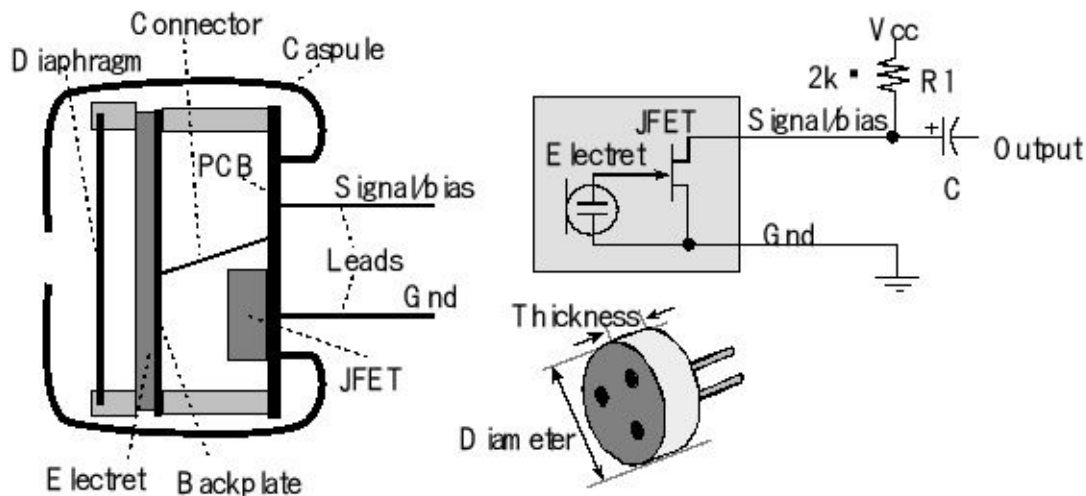
The two sets of data, together, have  $2N-2$  degrees of freedom. If  $N=10$ , the number in the  $df=18$  row, **confidence**=99% column is 2.878. This means if  $H_0$  is true, then the probability that  $t$  is less than  $-2.878 = 0.005$  and the probability that  $t$  is greater than  $2.878 = 0.005$ . Therefore, the probability of  $-2.878 < t < 2.878 = 0.99$  (confidence interval of 99%)

If we collect data and calculate  $t$  such that the test statistic  $t$  is greater than 2.878 or less than  $-2.878$ , then we claim “we reject the hypothesis  $H_0$ ”. If the test statistic  $t$  is between  $-2.878$  and  $2.878$  we do not claim the hypothesis to be true. In other words, we have not proven the means to be equal. Rather, we say “we do not reject the hypothesis  $H_0$ ”. If  $t$  is greater than 2.878 or less than  $-2.878$ , then we claim the resolution of the system is less than or equal to  $\Delta x$ .

### 5.1.3. Audio Input/Output

A **microphone** is a type of displacement transducer. Sound waves, which are pressure waves travelling in air, cause a diaphragm to vibrate, and the diaphragm

motion causes the distance between capacitor plates to change. This variable capacitance creates a voltage, which can be amplified and recorded. The **electret condenser microphone** (ECM) is an inexpensive choice for converting sound to analog voltage. Electret microphones are used in consumer and communication audio devices because of their low cost and small size. For applications requiring high sensitivity, low noise, and linear response, we could use the dynamic microphone, like the ones used in high-fidelity audio recording equipment. The ECM capsule acts as an acoustic resonator for the capacitive electret sensor shown in Figure 5.6. The ECM has a **Junction Field Effect Transistor** (JFET) inside the transducer providing some amplification. This JFET requires power as supplied by the R1 resistor. This local amplification allows the ECM to function with a smaller capsule than typically found with other microphones. ECM devices are cylindrically shaped, have a diameter ranging from 3 to 10 mm, and have a thickness ranging from 1 to 5 mm.



*Figure 5.6. Physical and electrical view of an ECM with JFET buffer ( $V_{cc}$  depends on microphone)*

An ECM consists of a pre-charged, non-conductive membrane between two plates that form a capacitor. The backplate is fixed, and the other plate moves with sound pressure. Movement of the plate results in a capacitance change, which in turn results in a change in voltage due to the non-conductive, pre-charged membrane. An electrical representation of such an acoustic sensor consists of a signal voltage source in series with a source capacitor. The most common method of interfacing this sensor is a high-impedance buffer/amplifier. A single JFET with its gate connected to the sensor plate and biased as shown in Figure 5.7 provides buffering and amplification. The capacitor C provides high-pass filtering, so the voltage at the output will be less than  $\pm 100$  mV for normal voice. Audio microphones need additional amplification and band-pass filtering. Typical audio signals exist from 100 Hz to 10 kHz. The presence of the R1 resistor is called "phantom biasing". The electret has two connections: Gnd and Signal/bias. Typically, the metallic capsule is connected to Gnd.

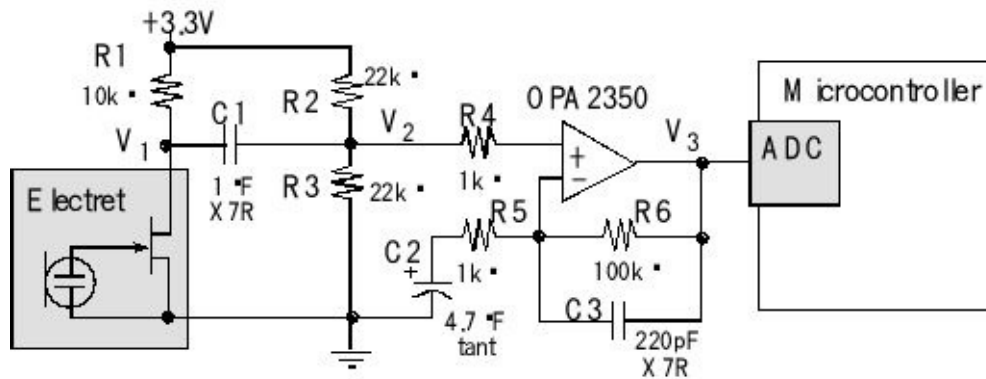


Figure 5.7. An electret microphone can be used to record sound.

Many electret data sheets suggest an  $R1$  of 2 k $\Omega$ , but signal-to-noise ratio can be improved by using a 10 k $\Omega$  resistor. The series capacitor  $C1$  creates a high pass filter. Because the output of a high pass filter would normally include positive and negative voltages, we will need a way to offset the circuit so all voltages exist from 0 to +3.3 V, allowing the use of a single supply and rail-to-rail op amps.  $R2$  and  $R3$  provide an offset for the high pass filter, so the signal  $V2$  will be the sound signal plus a fixed offset of 1.65 V. The effective impedance from  $V2$  to ground is 11 k $\Omega$ , so the HPF cutoff is  $1/(2\pi \cdot 0.22\mu\text{F} \cdot 11\text{k}\Omega) = 66$  Hz. The gain of the system is  $1+R6/R5$ , which will be 101. The capacitor  $C2$  will make the signal  $V3$  be the amplified sound plus 1.65 V. The gain is selected so the  $V3$  signal is  $1.65 \pm 1$  V for the sounds we wish to record. The capacitor  $C3$  provides a little low pass filtering, causing the amplifier gain to drop to one for frequencies above  $1/(2\pi \cdot 220\text{pF} \cdot 100\text{k}\Omega) = 7.2$  kHz. A better LPF would be to add an active LPF. The active LPF would also need a 1.65 V offset. If we wish to process sound with frequency components from 100 to 5 kHz, then we should sample at or above 10 kHz. The analog system must pass the signals of interest, but reject signals above  $\frac{1}{2}$  the sampling rate. One of the cost savings tradeoffs is to use a less analog filter (fewer poles) and increase the sampling rate, adding digital filtering. If we sampled sound with a 12-bit ADC, we should select a 12-bit DAC to output the sound. We could improve signal to noise by replacing the +3.3 V connected to  $R1$  and  $R2$  in Figure 5.7 with a LM4041 adjustable reference and create a low noise 3.0V voltage.

The LM4041CILP is a shunt reference used to make the analog reference required by the MAX5353 12-bit DAC. This DAC was previously interfaced in Example 7.2 of Volume 2. The MC34119 audio amp can be used to amplify the DAC output providing the current needed to drive a typical 8- $\Omega$  speaker (Figure 5.8). The gain of the audio amplifier is  $2 \cdot R11/R10$ , which for this circuit will be one. This means a 2-V peak-to-peak signal out of the DAC will translate to a 2-V peak-to-peak signal on the speaker. The maximum power that the MC34119 can deliver to the speaker is 250 mW, so the software should limit the sound signal below 1.4 V<sub>rms</sub> when driving an 8- $\Omega$  speaker. The quality of sound can be increased by selecting a better speaker and placing the speaker into an enclosure. For more information on how to design a

speaker box, perform a web search on “speaker enclosure”.

Software in Program 7.2 (Volume 2) can be used to interface the MAX5353 12-bit DAC. Program 5.1 performs the sound input and output. The sampling rate is 10 kHz. The ADC code was presented earlier in Chapter 2.

```

void ADC3_Handler(void){ int16_t data;
  ADC_ISC_R = ADC_ISC_IN3; // acknowledge ADC sequence 3 completion
  data = (ADC_SSFIFO3_R&ADC_SSFIFO3_DATA_M)-512; // 10-bit sound
// process, filter, record etc.
  DAC_Out(data);
}
void main(void){
  PLL_Init(); // now running at 80 MHz
  ADC_InitTimer0ATriggerSeq3PD3(7999); // sample at 10 kHz
  DAC_Init(2048); // Volume 2, Program 7.2
  while(1){ };
}

```

Program 5.1. Real-time sound output input/output.

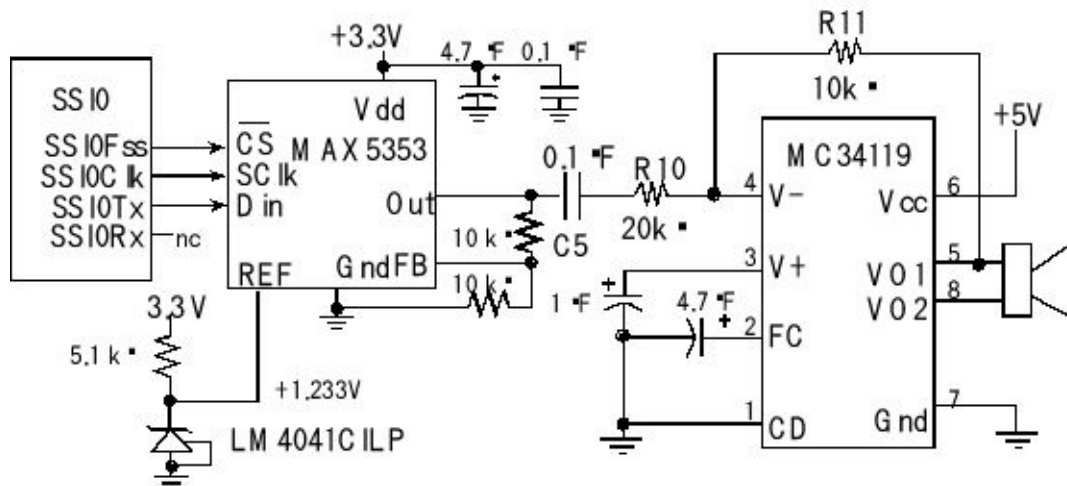


Figure 5.8. A DAC and an audio amplifier allow the microcontroller to output sound.

## 5.2. Priority scheduler

### 5.2.1. Implementation

To implement priority, we add another field to the TCB, see Program 5.2. In this system we define 0 as the highest priority and 254 as the lowest. In some operating systems, each thread must have unique priority, but in this chapter multiple threads can have the same priority. If we have multiple threads with equal priority, these threads will be run in a round robin fashion. The strategy will be to find the highest priority thread, which is neither blocked nor sleeping and run it as shown in Figure 5.9.

```
struct tcb{
  int32_t *sp;    // pointer to stack (valid for threads not running)
  struct tcb *next; // linked-list pointer
  int32_t *BlockPt; // nonzero if blocked on this semaphore
  uint32_t Sleep; // nonzero if this thread is sleeping
  uint8_t Priority; // 0 is highest, 254 lowest
};
```

Program 5.2. TCB for the priority scheduler.

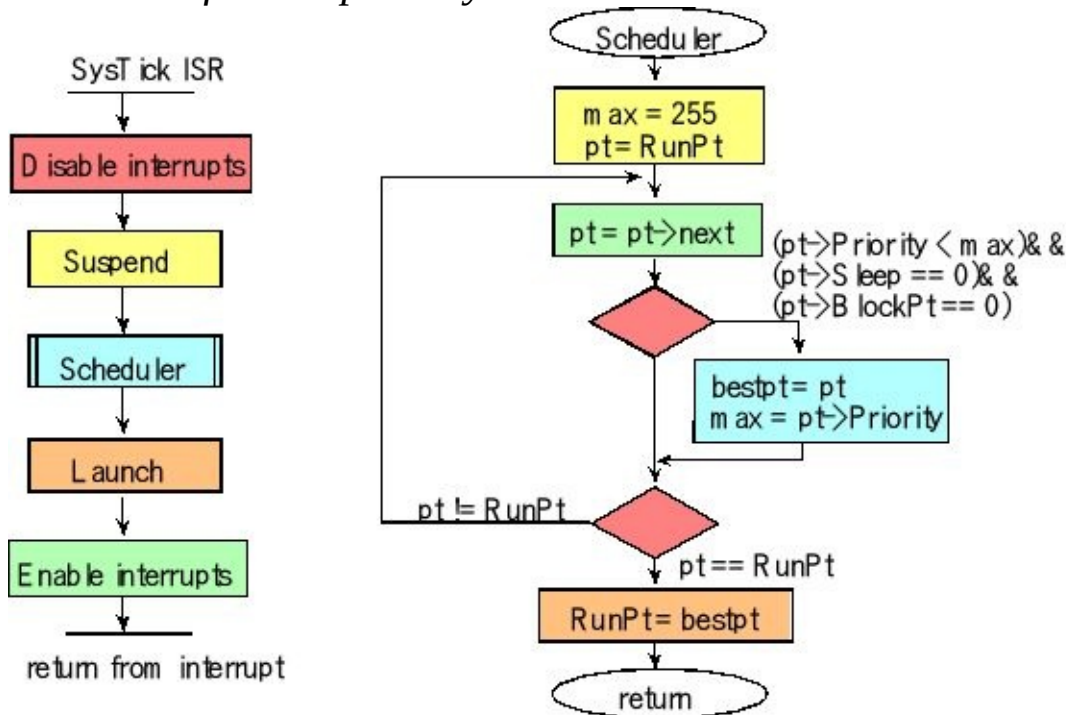


Figure 5.9. Priority scheduler finds the highest priority thread.

**Observation:** Normally, we add priority to a system that implements blocking semaphores and not to one that uses spinlock semaphores.

If there are multiple threads at that highest priority that are not sleeping nor blocked, then the scheduler will run them in a round robin fashion. The statement, `pt = pt->next` guarantees that the same higher priority task is not picked again.

```
void Scheduler(void){ // every time slice
    uint32_t max = 255; // max
    tcbType *pt;
    tcbType *bestPt;
    pt = RunPt; // search for highest thread not blocked or sleeping
    do{
        pt = pt->next; // skips at least one
        if((pt->Priority < max)&&((pt->BlockPt)==0)&&((pt->Sleep)==0)){
            max = pt->Priority;
            bestPt = pt;
        }
    } while(RunPt != pt); // look at all possible threads
    RunPt = bestPt;
}
```

*Program 5.3. One possible priority scheduler.*

**Checkpoint 5.1:** If there are  $N$  threads in the TCB list, how many threads must the scheduler in Program 5.3 consider before choosing the thread the next thread to run? In other words, how many times does the do-while loop run?

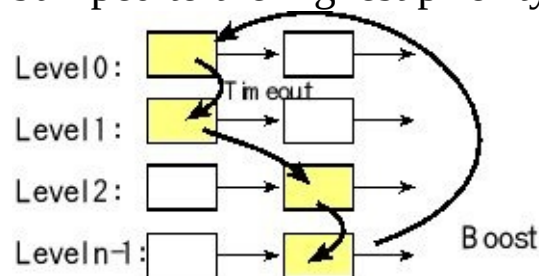
There are many approaches to assigning priority. If the system is **I/O centric** then we can assign high priority to I/O bound threads and low priority to CPU bound threads. Another approach is to define a cost to various performance metrics like lateness, and bandwidth, and then assign priorities that **minimize cost**. A dynamic scheduler is one that adjusts priority at run time. Examples include **earliest deadline first** (EDF) and **least slack time first** (LST) (EDF)

## 5.2.2. Multi-level Feedback Queue

The priority scheduler in the previous section will be inefficient if there are a lot of threads. Because the scheduler must look at all threads, the time to run the scheduler grows linearly with the number of threads. One implementation that is appropriate for priority systems with many threads is called the multi-level feedback queue (MLFQ). MLFQ was introduced in 1962 by Corbato et al. and has since been adopted in some form by all the major operating systems, BSD Unix and variants, Solaris and Windows. Its popularity stems from its ability to optimize performance with respect to two metrics commonly used in traditional Operating Systems. These metrics are *turnaround time*, and *response time*. Turnaround time is the time elapsed from when a thread arrives till it completes execution. Response time is the time elapsed from when a thread arrives till it starts execution. Let  $S$  be the average time to service a

request, and  $R$  be the average response time (waiting+service). One nondimensional metric for response time is **normalized mean response time**,  $R/S$ . Preemptive scheduling mechanisms like Shortest Time-to-completion First (STCF) and Round-Robin (RR) are optimal at minimizing the average turnaround time and response time respectively. However, both perform well on only one of these metrics and show very poor performance with respect to the other. MLFQ fairs equally well on both these metrics. As the name indicates, MLFQ has multiple queues, one per priority level, with multiple threads operating at the same priority level. In keeping with our description of priority, we assume level 0 is the highest priority and higher levels imply lower priority. There will be a finite number of priority levels from 0 to  $n-1$ , see Figure 5.10. The rules that govern the processing of these queues by the scheduler are as follows:

1. Startup: All threads start at the highest priority. Start in queue at level 0.
2. Highest runs: If  $\text{Priority}(T_i) < \text{Priority}(T_j)$  then  $T_i$  is scheduled to run before  $T_j$ .
3. Equals take turns: If  $\text{Priority}(T_i) = \text{Priority}(T_j)$  then  $T_i$  and  $T_j$  are run in RR order.
4. True accounting: If a thread uses up its *timeslice* at priority  $m$  then its priority is reduced to  $m+1$ . It is moved to the corresponding queue.
5. Priority Boost: The scheduler does a periodic reset, where all threads are bumped to the highest priority.



*Figure 5.10. The shaded task in this figure begins in the level 0 (highest) priority queue. If it runs to the end of its 10-ms time slice (timeout), it is bumped to level 1. If it again runs to the end of its 10-ms time slice, it is bumped to level 2. Eventually, a thread that does not sleep or block will end up in the lower priority queue. Periodically the system will reset and place all threads back at level 0.*

An obvious precondition to choosing a thread is to make sure it is “ready”, that is, it is not blocked on a resource or sleeping. This rule is implicit and hence not listed here. Rules 2, and 3 are self-explanatory as MLFQ attempts to schedule the highest priority ready thread at any time. Rule 1 makes sure that every thread gets a shot at



executing as quickly as possible, the first time it enters the system. Rule 4 is what determines when a thread is moved from one level to another. Further, whether a thread uses up its timeslice at one shot or over multiple runs, true accounting requires that the accumulated time for the thread at a given priority level be considered. There are versions of MLFQ that let a thread remain at a priority level with its accrued time towards the timeslice reset to zero, if it blocked on a resource. These versions allowed the possibility of gaming the scheduler. Without rule 5, MLFQ eventually reduces to RR after running for a while with all threads operating at the lowest priority level. By periodically boosting all threads to the highest priority, rule 5 causes a scheduler reset that lets the scheduler adapt to changes in thread behavior.

### 5.2.3. Starvation and aging

One disadvantage of a priority scheduler on a busy system is that low priority threads may never be run. This situation is called **starvation**. For example, if a high priority thread never sleeps or blocks, then the lower priority threads will never run. It is the responsibility of the user to assign priorities to tasks. As mentioned earlier, as processor utilization approaches one, there will not be a solution. In general, starvation is not a problem of the RTOS but rather a result of a poorly designed user code.

One solution to starvation is called **aging**. In this scheme, threads have a permanent fixed priority and a temporary working priority, see Program 5.4. The permanent priority is assigned according to the rules of the previous paragraph, but the temporary priority is used to actually schedule threads. Periodically the OS increases the temporary priority of threads that have not been run in a long time. For example, the **Age** field is incremented once every 1ms if the thread is not blocked or not sleeping. For every 10 ms the thread has not been run, its **WorkingPriority** is reduced. Once a thread is run, its temporary priority is reset back to its permanent priority. When the thread is run, the **Age** field is cleared and the **FixedPriority** is copied into the **WorkingPriority**.

```
struct tcb{
  int32_t *sp;    // pointer to stack (valid for threads not running)
  struct tcb *next; // linked-list pointer
  int32_t *BlockPt; // nonzero if blocked on this semaphore
  uint32_t Sleep; // nonzero if this thread is sleeping
  uint8_t WorkingPriority; // used by the scheduler
  uint8_t FixedPriority; // permanent priority
  uint32_t Age; // time since last execution
};
```

*Program 5.4. TCB for the priority scheduler.*

## 5.2.4. Priority inversion and inheritance on Mars Pathfinder

Another problem with a priority scheduler is **priority inversion**, a condition where a high-priority thread is waiting on a resource owned by a low-priority thread. For example, consider the case where both a high priority and low priority thread need the same resource. Assume the low-priority thread asks for and is granted the resource, and then the high-priority thread asks for it and blocks. During the time the low priority thread is using the resource, the high-priority thread essentially becomes low priority. The scenario in Figure 5.11 begins with a low priority meteorological task asking for and being granted access to a shared memory using the **mutex** semaphore. The second step is a medium priority communication task runs for a long time. Since communication is higher priority than the meteorological task, the communication task runs but the meteorological task does not run. Third, a very high priority task starts but also needs access to the shared memory, so it calls `wait(&mutex)`. This high priority task, however, will block because **mutex** is 0. Notice that while the communication task is running, this high priority task effectively runs at low priority because it is blocked on a semaphore captured previously by the low priority task.

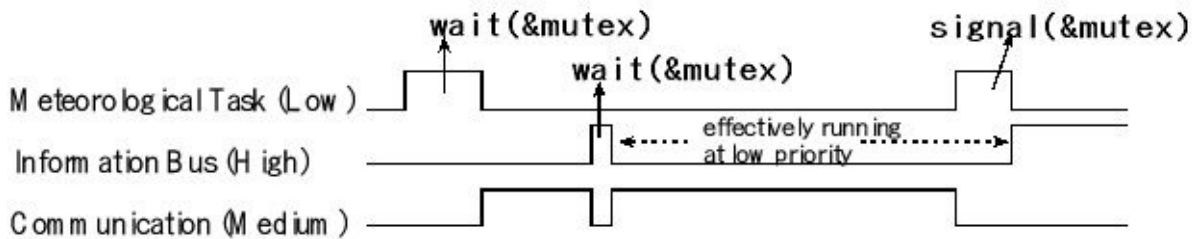


Figure 5.11. Priority inversion as occurred with Mars Pathfinder.

[http://research.microsoft.com/en-us/um/people/mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html)

One solution to priority inversion is **priority inheritance**. With priority inheritance, once a high-priority thread blocks on a resource, the thread holding that resource is granted a temporary priority equal to the priority of the high-priority blocked thread. Once the thread releases the resource, its priority is returned to its original value.

A second approach is called **priority ceiling**. In this protocol each semaphore is assigned a priority ceiling, which is a priority equal to the highest priority of any task which may block on a semaphore for that resource. With priority ceiling, once a high-priority thread blocks on a resource, the thread holding that resource is granted a temporary priority equal to the priority of the priority ceiling. Just like inheritance, once the thread releases the resource, its priority is returned to its original value.

---

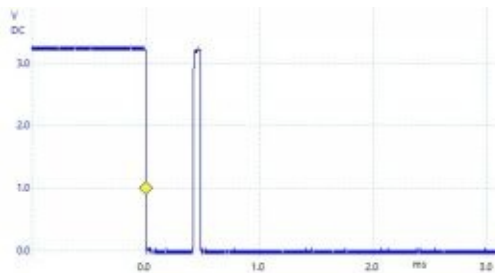
## 5.3. Debouncing a switch

### 5.3.1. Approach to debouncing

One of the problems with switches is called **switch bounce**. Many inexpensive switches will mechanically oscillate for up to a few milliseconds when touched or released. It behaves like an underdamped oscillator. These mechanical oscillations cause electrical oscillations such that a port pin will oscillate high/low during the bounce.

Contact bounce is a typical problem when interfacing switches. Figure 5.12 shows an actual voltage trace occurring when a negative logic switch is touched. On both a touch and release, there can be from 0 to 2 ms of extra edges, called switch bounce. However, sometimes there is no bounce.

This bounce is a problem when the system uses the switch to trigger important events. There are two problems to solve: 1) remove the bounce so there is one software event attached to the switch touch; 2) remove the bounce in such a way that there is low latency between the physical touch and the execution of the associated software task.



*Figure 5.12. Because of the mass and spring some switches bounce.*

In some cases, this bounce should be removed. To remove switch bounce we can ignore changes in a switch that occur within 10 ms of each other. In other words, recognize a switch transition, disarm interrupts for 10ms, and then rearm after 10 ms.

Alternatively, we could record the time of the switch transition. If the time between this transition and the previous transition is less than 10ms, ignore it. If the time is more than 10 ms, then accept and process the input as a real event.

Another method for debouncing the switch is to use a periodic interrupt with a period greater than the bounce, but less than the time the switch is held down. Each interrupt we read the switch, if the data is different from the previous interrupt the software recognizes the switch event.

**Checkpoint 5.2:** Consider the periodic interrupt method for debouncing a switch. Assume the interrupt period is 20 ms. What are the maximum and average

latencies (time between switch touch and execution of the task)?

### 5.3.2. Debouncing a switch on TM4C123

If we have a RTOS we can use a semaphore to debounce a switch. In order to run the user task immediately on touch we will configure the GPIO input to trigger an interrupt on both edges. However, there can be multiple falling and rising edges on both a touch and a release, see Figure 5.13. A main thread will wait on that semaphore, sleep for 10ms and then read the switch. The interrupt occurs at the start of the bouncing, but the reading of the switch occurs at a time when the switch state is stable. We will disarm the interrupt during the ISR, so the semaphore is incremented once per touch and once per release. We will rearm the interrupt at the stable time. Program 5.5 and Figure 5.14 show one possible solution that executes **Touch1** when the switch SW1 is touched, and it executes **Touch2** when switch SW2 is touched.

We can set the priorities of the hardware interrupt and main threads depending on the importance of the software event. If the edge-triggered interrupt has high priority, the semaphore will be signaled immediately after a hardware touch/release event. Furthermore, the main threads also have high priority, the software responses will also be run immediately. Notice the **OS\_Suspend()** call at the end of the ISR. This will run the scheduler.

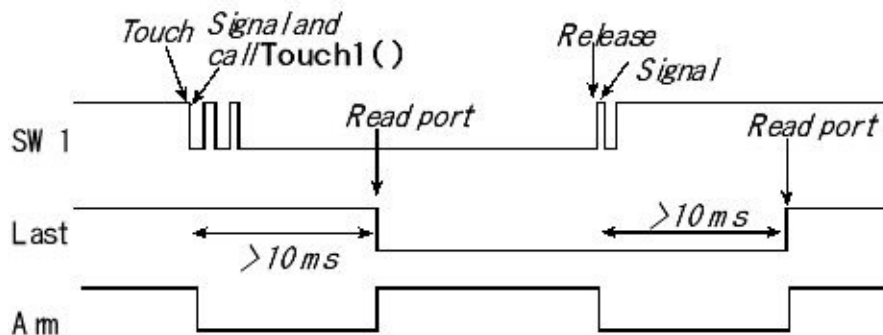


Figure 5.13. Touch and release both cause the ISR to run. The port is read during the stable time

```
int32_t SW1,SW2;
uint8_t last1,last2;
void Switch_Init(void){
    SYSCTL_RCGCGPIO_R |= 0x20;    // activate clock for Port F
    OS_InitSemaphore(&SW1,0);    // initialize semaphores
    OS_InitSemaphore(&SW2,0);
    GPIO_PORTF_LOCK_R = 0x4C4F434B; // unlock GPIO Port F
    GPIO_PORTF_CR_R = 0x1F;    // allow changes to PF4-0
    GPIO_PORTF_DIR_R &= ~0x11; // make PF4,PF0 in
    GPIO_PORTF_DEN_R |= 0x11; // enable digital I/O on PF4,PF0
    GPIO_PORTF_PUR_R |= 0x11; // pullup on PF4,PF0
```

```

GPIO_PORTF_IS_R &= ~0x11;    // PF4,PF0 are edge-sensitive
GPIO_PORTF_IBE_R |= 0x11;    // PF4,PF0 are both edges
GPIO_PORTF_ICR_R = 0x11;    // clear flags
GPIO_PORTF_IM_R |= 0x11;    // arm interrupts on PF4,PF0
NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00200000; // priority 1
NVIC_EN0_R = 0x40000000;    // enable interrupt 30 in NVIC
}
void GPIOPortF_Handler(void){
if(GPIO_PORTF_RIS_R&0x10){ // poll PF4
    GPIO_PORTF_ICR_R = 0x10; // acknowledge flag4
    OS_Signal(&SW1);        // signal SW1 occurred
    GPIO_PORTF_IM_R &= ~0x10; // disarm interrupt on PF4
}
if(GPIO_PORTF_RIS_R&0x01){ // poll PF0
    GPIO_PORTF_ICR_R = 0x01; // acknowledge flag0
    OS_Signal(&SW2);        // signal SW2 occurred
    GPIO_PORTF_IM_R &= ~0x81; // disarm interrupt on PF0
}
OS_Suspend();}
void Switch1Task(void){ // high priority main thread
last1 = GPIO_PORTF_DATA_R&0x10;
while(1){
    OS_Wait(&SW1); // wait for SW1 to be touched/released
    if(last1){ // was previously not touched
        Touch1(); // user software associated with touch
    }else{
        Release1(); // user software associated with release
    }
    OS_Sleep(10); // wait for bouncing to be over
    last1 = GPIO_PORTF_DATA_R&0x10;
    GPIO_PORTF_IM_R |= 0x10; // rearm interrupt on PF4
    GPIO_PORTF_ICR_R = 0x10; // acknowledge flag4
}
}
void Switch2Task(void){ // high priority main thread
last2 = GPIO_PORTF_DATA_R&0x01;
while(1){
    OS_Wait(&SW2); // wait for SW2 to be touched/released
    if(last2){ // was previously not touched
        Touch2(); // user software associated with touch
    }else{
        Release2(); // user software associated with release
    }
    OS_Sleep(10); // wait for bouncing to be over
}
}

```

```

last2 = GPIO_PORTF_DATA_R&0x01;
GPIO_PORTF_IM_R |= 0x01; // rearm interrupt on PF0
GPIO_PORTF_ICR_R = 0x01; // acknowledge flag0
}
}

```

Program 5.5. Interrupt-driven edge-triggered input that calls *Touch1()* on the falling edge of PF4, calls *Release1()* on the rising edge of PF4, calls *Touch2()* on the falling edge of PF0 and calls *Release2()* on the rising edge of PF0.

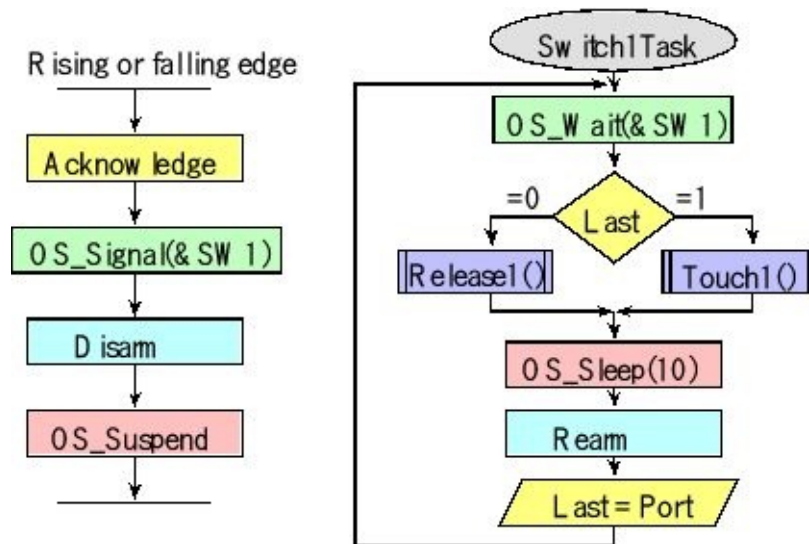


Figure 5.14. Flowchart of a RTOS-solution to switch bounce. *Switch1Task* is a high-priority main thread. Notice that *Release1* is executed immediately after a release, and *Touch1* is executed immediately after the switch is touched. However the global variable *Last* is set at a time the switch is guaranteed to be stable.

### 5.3.3. Debouncing a switch on MSP432

If we have a RTOS we can perform a similar sequence. In particular, we will use Program 5.6 to signal a semaphore. Even though we armed the interrupt for fall, there can be multiple falling edges on both a touch and a release. A high priority main thread will wait on that semaphore, sleep for 10ms and then read the switch. The interrupt occurs at the start of the bouncing, but the reading of the switch occurs at a time when the switch state is stable. We will disarm the interrupt during the ISR, so the semaphore is incremented once per touch or once per release. We will rearm the interrupt at the stable time. Program 5.6 and Figure 5.15 show one possible solution that executes *Touch1* when the switch SW1 is touched, and it executes *Touch2* when switch SW2 is touched.

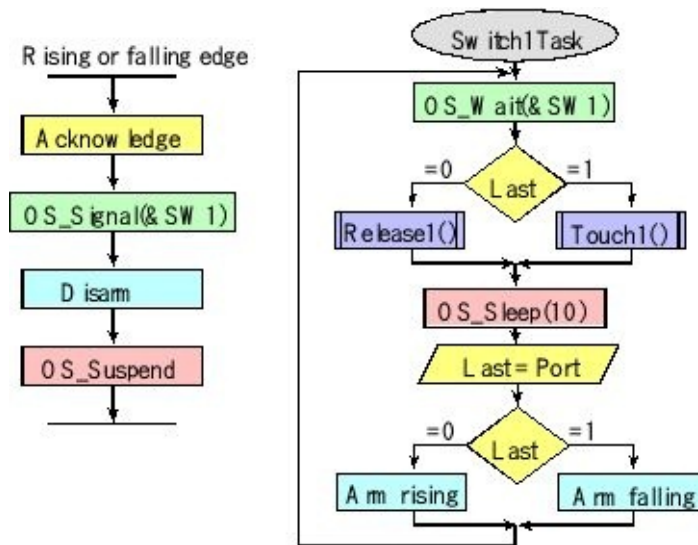


Figure 5.15. Flowchart of a RTOS-solution to switch bounce. Switch1Task is a high-priority main thread. Notice that Release1 is executed immediately after a release, and Touch1 is executed immediately after the switch is touched. However, the global variable Last is set at a time the switch is guaranteed to be stable.

```

int32_t SW1,SW2;
uint8_t last1,last2;
void Switch_Init(void){
    SW1 = SW2 = 0;           // initialize semaphores
    P1SEL1 &= ~0x12;        // configure P1.1, P1.4 as GPIO
    P1SEL0 &= ~0x12;        // built-in Buttons 1 and 2
    P1DIR &= ~0x12;         // make P1.1, P1.4 in
    P1REN |= 0x12;          // enable pull resistors
    P1OUT |= 0x12;          // P1.1, P1.4 is pull-up
    P1IES |= 0x12;          // P1.1, P1.4 is falling edge event
    P1IFG &= ~0x12;         // clear flag1 and flag4
    P1IE |= 0x12;           // arm interrupt on P1.1, P1.4
    NVIC_IPR8 = (NVIC_IPR8&0x00FFFFFF)|0x20000000; // (f) priority 1
    NVIC_ISER1 = 0x00000008; // enable interrupt 35 in NVIC
void PORT1_IRQHandler(void){ uint8_t status;
    status = P1IV; // 4 for P1.1 and 10 for P1.4
    if(status == 4){
        OS_Signal(&SW1); // SW1 occurred
        P1IE &= ~0x02; // disarm interrupt on P1.2
    }
    if(status == 10){
        OS_Signal(&SW2); // SW2 occurred
        P1IE &= ~0x10; // disarm interrupt on P1.4
        OS_Suspend();
    }
void Switch1Task(void){ // high priority main thread
    last1 = P1IN&0x02;

```

```

while(1){
    OS_Wait(&SW1); // wait for SW1 to be touched/released
    if(last1){    // was previously not touched
        Touch1(); // user software associated with touch
    }else{
        Release1(); // user software associated with release
    }
    OS_Sleep(10);
    last1 = P1IN&0x02;
    if(last1){
        P1IES |= 0x02; // next will be falling edge
    }else{
        P1IES &= ~0x02; // next will be rising edge
    }
    P1IE |= 0x02; // rearm interrupt on P1.1
    P1IFG &= ~0x02; // clear flag1
}
}
void Switch2Task(void){ // high priority main thread
    last2 = P1IN&0x10;
    while(1){
        OS_Wait(&SW2); // wait for SW2 to be touched/released
        if(last2){    // was previously not touched
            Touch2(); // user software associated with touch
        }else{
            Release2(); // user software associated with release
        }
        OS_Sleep(10);
        last2 = P1IN&0x10;
        if(last2){
            P1IES |= 0x10; // next will be falling edge
        }else{
            P1IES &= ~0x10; // next will be rising edge
        }
        P1IE |= 0x10; // rearm interrupt on P1.4
        P1IFG &= ~0x10; // clear flag4
    }
}
}

```

*Program 5.6. Interrupt-driven edge-triggered input that calls Touch1() on the falling edge of P1.1, calls Release1() on the rising edge of P1.1, calls Touch2() on the falling edge of P1.4 and calls Release2() on the rising edge of P1.4.*



---

## 5.4. Running event threads as high priority main threads

In the previous chapters, we ran time-critical tasks (event tasks) directly from the interrupt service routine. Now that we have a priority scheduler, we can place time-critical tasks as high priority main threads. We will block these time-critical tasks waiting on an event (semaphore), and when the event occurs we signal its semaphore. Because we now have a high priority thread not blocked, the scheduler will run it immediately. In Program 5.7, we have a periodic interrupt that simply signals a semaphore and invokes the scheduler. If we assign the program **Task0** as a high priority main thread, it will be run periodically with very little jitter.

It may seem like a lot of trouble to run a periodic task. One might ask why not just put the time-critical task in the interrupt service routine. A priority scheduler is flexible in two ways. First, because it implements priority we can have layers of important, very important and very very important tasks. Second, we can use this approach for any triggering event, hardware or software. We simply make that triggering event call `OS_Signal` and `OS_Suspend`. One of the advantages of this approach is the separation of the user/application code from the OS code. The OS simply signals the semaphore on the appropriate event and the user code runs as a main thread.

```
int32_t TakeSoundData; // binary semaphore
void RealTimeEvents(void){
    OS_Signal(&TakeSoundData);
    OS_Suspend();
}
void Task0(void){
    while(1){
        OS_Wait(&TakeSoundData); // signaled every 1ms
        TExaS_Task0(); // toggle virtual logic analyzer
        Profile_Toggle0(); // viewed by logic analyzer to know Task0 started
// time-critical software
    }
}
int main(void){
    OS_Init();
// other initialization
    OS_InitSemaphore(&TakeSoundData,0);
    OS_AddThreads(&Task0,0,&Task1,1,&Task2,2, &Task3,3,
        &Task4,3, &Task5,3, &Task6,3, &Task7,4);
    BSP_PeriodicTask_InitC(&RealTimeEvents,1000,0);
    TExaS_Init(LOGICANALYZER, 1000); // initialize the logic analyzer
```

```
OS_Launch(BSP_Clock_GetFreq()/THREADFREQ); // doesn't return  
return 0; // this never executes  
}
```

*Program 5.7. Running time-critical tasks as high priority event threads.*

---

## 5.5. Available RTOS

### 5.5.1. Micrium $\mu$ C/OS-II

We introduced several concepts that common in real-time operating systems but ones we don't implement in our simple RTOS. To complete this discussion, we explore some of the popular RTOSs (for the ARM Cortex-M) in commercial use and how they implement some of the features we covered.

Micrium  $\mu$ C/OS-II is a portable, ROMable, scalable, preemptive, real-time deterministic multitasking kernel for microprocessors, microcontrollers and DSPs (for more information, see <http://micrium.com/rtos/ucosii/overview/>). **Portable** means user and OS code written on one processor can be easily shifted to another processor. **ROMable** is a standard feature of most compilers for embedded systems, meaning object code is programmed into ROM, and variables are positioned in RAM. **Scalable** means applications can be developed on this OS for 10 threads, but the OS allows expansion to 255 threads. Like most real-time operating systems, high priority tasks can preempt lower priority tasks. Because each thread in Micrium  $\mu$ C/OS-II has a unique priority (no two threads have equal priority), the threads will run in a deterministic pattern, making it easy to certify performance. In fact, the following lists the certifications available for Micrium  $\mu$ C/OS-II

- MISRA-C:1998
- DO178B Level A and EUROCAE ED-12B
- Medical FDA pre-market notification (510(k)) and pre-market approval (PMA)
- SIL3/SIL4 IEC for transportation and nuclear systems
- IEC-61508

As of December 2016, Micrium  $\mu$ C/OS-II is available for over 50 processor architectures, including the Cortex M3 and Cortex M4F. Ports are available for download on <http://micrium.com>. Micrium  $\mu$ C/OS-II manages up to 255 application tasks.  $\mu$ C/OS-II includes: semaphores; event flags; mutual-exclusion semaphores that eliminate unbounded priority inversions; message mailboxes and queues; task, time and timer management; and fixed sized memory block management.

Micrium  $\mu$ C/OS-II's footprint can be scaled (between 5 kibibytes to 24 kibibytes) to only contain the features required for a specific application. The execution time for most services provided by  $\mu$ C/OS-II is both constant and deterministic; execution times do not depend on the number of tasks running in the application.

To provide for stability and protection, this OS runs user code with the PSP and OS

code with the MSP. The way in which the Micrium  $\mu$ C/OS supports many processor architectures is to be layered. Only a small piece of the OS code is processor specific. It also provides a **Board Support Package (BSP)** so the user code can also be layered, see Figure 5.16.

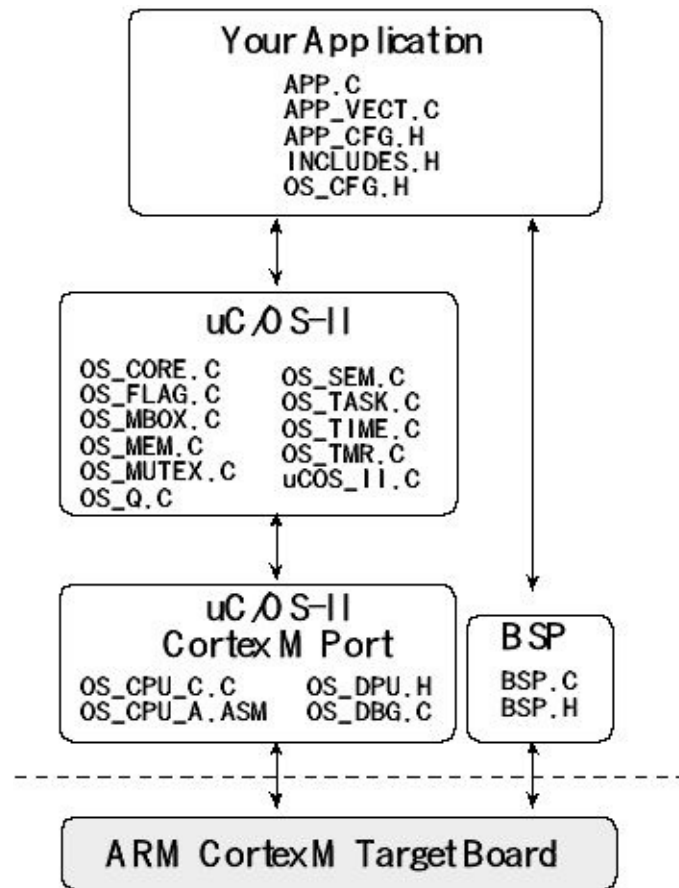


Figure 5.16. Block diagram of the Micrium  $\mu$ C/OSII.

To illustrate the operation of Micrium  $\mu$ C/OS-II, Program 5.8 shows the thread-switch code. PendSV is an effective method for performing context switches with Cortex-M because the Cortex-M saves R0-R3, R12, LR, PC, PSW on any exception, and restores the same on return from exception. So only saving of R4-R11 is required and fixing up the stack pointers. Using the PendSV exception this way means that context saving and restoring is identical whether it is initiated from a thread or occurs due to an interrupt or exception. On entry into PendSV handler 1) xPSR, PC, LR, R12, R0-R3 have been saved on the process stack (by the processor); 2) Processor mode is switched to Handler mode (from Thread mode); 3) The stack is now the Main stack (switched from Process stack); 3) **OSTCBCur** points to the **OS\_TCB** of the task to suspend; and 4) **OSTCBHighRdy** points to the **OS\_TCB** of the task to resume. There nine steps for switching a thread:

1. Get the process SP, if 0 then go to step 4. the saving part (first switch);
2. Save remaining regs R4-R11 on process stack;
3. Save the process SP in its TCB, **OSTCBCur->OSTCBStkPtr = SP;**
4. Call **OSTaskSwHook();**
5. Get current high priority, **OSPrioCur = OSPrioHighRdy;**

6. Get current ready thread TCB, **OSTCBCur = OSTCBHighRdy;**
7. Get new process SP from TCB, **SP = OSTCBHighRdy->OSTCBStkPtr;**
8. Restore R4-R11 from new process stack;
9. Perform exception return which will restore remaining context.

### **OS\_CPU\_PendSVHandler**

```

CPSID I ; Prevent interruption during context switch
MRS R0, PSP ; PSP is process stack pointer
CBZ R0, OS_CPU_PendSVHandler_nosave ; Skip first time
SUBS R0, R0, #0x20 ; Save remaining regs R4-11 on process stack
STM R0, {R4-R11}
LDR R1, =OSTCBCur ; OSTCBCur->OSTCBStkPtr = SP;
LDR R1, [R1]
STR R0, [R1] ; R0 is SP of process being switched out

```

**; At this point, entire context of process has been saved**

### **OS\_CPU\_PendSVHandler\_nosave**

```

PUSH {R14} ; Save LR exc_return value
LDR R0, =OSTaskSwHook ; OSTaskSwHook();
BLX R0
POP {R14}
LDR R0, =OSPrioCur ; OSPrioCur = OSPrioHighRdy;
LDR R1, =OSPrioHighRdy
LDRB R2, [R1]
STRB R2, [R0]
LDR R0, =OSTCBCur ; OSTCBCur = OSTCBHighRdy;
LDR R1, =OSTCBHighRdy
LDR R2, [R1]
STR R2, [R0]
LDR R0, [R2] ; R0 is new PSP; SP = OSTCBHighRdy->OSTCBStkPtr;
LDM R0, {R4-R11} ; Restore R4-11 from new process stack
ADDS R0, R0, #0x20
MSR PSP, R0 ; Load PSP with new process SP
ORR LR, LR, #0x04 ; Ensure exception return uses process stack
CPSIE I
BX LR ; Exception return will restore remaining context

```

*Program 5.8. Thread switch code on the Micrium uC/OSII.*

Since PendSV is set to lowest priority in the system, we know that it will only be run when no other exception or interrupt is active, and therefore safe to assume that context being switched out was using the process stack (PSP). Micrium  $\mu$ C/OS-II provides numerous hooks within the OS to support debugging, profiling, and feature expansion. An example of a hook is the call to **OSTaskSwHook()**. The user can

specify the action invoked by this call.

Micrium  $\mu$ C/OS-III extends this OS with many features as more threads, round-robin scheduling, enhanced messaging, extensive performance measurements, and time stamps.

## 5.5.2. Texas Instruments RTOS

TI-RTOS scales from a real-time multitasking kernel to a complete RTOS solution including additional middleware components and device drivers. TI-RTOS is provided with full source code and requires no up-front or runtime license fees. TI-RTOS Kernel is available on most TI microprocessors, microcontrollers and DSPs. TI-RTOS middleware, drivers and board initialization components are available on select ARM<sup>®</sup> Cortex<sup>™</sup>-M4 Tiva-C, C2000<sup>™</sup> dual core C28x + ARM Cortex-M3, MSP430 microcontrollers, and the SimpleLink<sup>™</sup> WiFi<sup>®</sup> CC3200. For more information, see <http://www.ti.com/tool/ti-rtos> or search RTOS on [www.ti.com](http://www.ti.com). TI-RTOS combines a real-time multitasking kernel with additional middleware components including TCP/IP and USB stacks, a FAT file system, and device drivers, see Figure 5.17 and Table 5.2. TI-RTOS provides a consistent embedded software platform across TI's microcontroller devices, making it easy to port legacy applications to the latest devices.

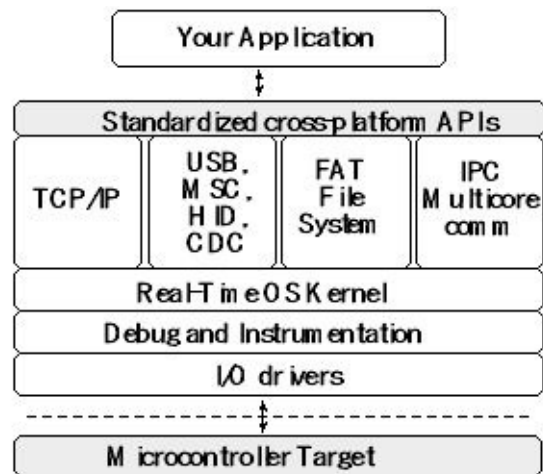


Figure 5.17. Block diagram of the Texas Instruments RTOS.

TI-RTOS Module	Description
TI-RTOS Kernel	TI-RTOS Kernel provides deterministic preemptive multithreading and synchronization services, memory management, and interrupt handling. TI-RTOS Kernel is highly scalable down to a few KBs of memory.
TI-RTOS Networking	TI-RTOS Networking provides an IPv4 and IPv6-compliant TCP/IP stack along with

	associated network applications such as DNS, HTTP, and DHCP.
TI-RTOS File System	TI-RTOS File System is a FAT-compatible file system based on the open source Fatfs product.
TI-RTOS USB	TI-RTOS USB provides both USB Host and Device stacks, as well as MSC, CDC, and HID class drivers. TI-RTOS USB uses the proven TivaWare USB stack.
TI-RTOS IPC	The TI-RTOS IPC provides efficient interprocessor communication in multicore devices.
TI-RTOS Instrumentation	TI-RTOS Instrumentation allows developers to include debug instrumentation in their application that enables run-time behavior, including context-switching, to be displayed by system-level analysis tools.
TI-RTOS Drivers and Board Initialization	TI-RTOS Drivers and Board Initialization provides a set of device driver APIs, such as Ethernet, UART and IIC, that are standard across all devices, as well as initialization code for all supported boards. All driver and board initialization APIs are built on the TivaWare, MWare, or MSP430Ware libraries.

**Table 5.2. Components of the TI RTOS.**

### 5.5.3. ARM RTX Real-Time Operating System

The Keil RTX is a royalty-free, deterministic Real-Time Operating System designed for ARM and Cortex-M devices. For more information, search RTX RTOS on **www.arm.com**. It allows you to create programs that simultaneously perform multiple functions and helps to create applications which are better structured and more easily maintained. RTX is available royalty-free and includes source code. RTX is deterministic. It has flexible scheduling including round-robin, pre-emptive, and collaborative. It operates at high speed with low interrupt latency. It has a small footprint. It supports unlimited number of tasks each with 254 priority levels. It provides an unlimited number of mailboxes, semaphores, mutex, and timers. It includes support for multithreading and thread-safe operation. There is debugging support in MDK-ARM. It has a dialog-based setup using  $\mu$ Vision Configuration Wizard.

RTX allows up to 250 active tasks. The priority scheduler supports up to 254 priority levels. The OS will dynamically check for valid stacks for running tasks. It implements timeouts, interval timing, and user timers. Synchronization and inter-task communication are handled by signals/events, semaphores, mutexes, and mailboxes. A task switch, the Cortex M3 version shown as Program 5.9, requires 192 bus cycles. The **STMDB** instruction saves the current thread and the **LDMIA** instruction restores the context for the next thread.

```

__asm void PendSV_Handler (void) {
    BL    __cpp(rt_pop_req) ; choose next thread to run
    LDR   R3,=__cpp(&os_tsk)
    LDM   R3,{R1,R2}      ; os_tsk.run, os_tsk.new
    CMP   R1,R2
    BEQ   Sys_Exit
    PUSH  {R2,R3}
    MOV   R3,#0
    STRB  R3,[R1,#TCB_RETUPD] ; os_tsk.run->ret_upd = 0
    MRS   R12,PSP          ; Read PSP
    STMDB R12!,{R4-R11}    ; Save Old context
    STR   R12,[R1,#TCB_TSTACK] ; Update os_tsk.run->tsk_stack
    BL    rt_stk_check      ; Check for Stack overflow
    POP   {R2,R3}
    STR   R2,[R3]          ; os_tsk.run = os_tsk.new
    LDR   R12,[R2,#TCB_TSTACK] ; os_tsk.new->tsk_stack
    LDMIA R12!,{R4-R11}    ; Restore New Context
    MSR   PSP,R12          ; Write PSP
    LDRB  R3,[R2,#TCB_RETUPD] ; Update ret_val?
    CBZ   R3,Sys_Exit
    LDRB  R3,[R2,#TCB_RETVAL] ; Write os_tsk.new->ret_val
    STR   R3,[R12]
    Sys_Exit MVN LR,#:NOT:0xFFFFFFFF ; set EXC_RETURN value
    BX   LR                ; Return to Thread Mode
}

```

*Program 5.9. Thread switch code on the ARM RTX RTOS (see file HAL\_CM3.c).*

ARM's Cortex Microcontroller Software Interface Standard (CMSIS) is a standardized hardware abstraction layer for the Cortex-M processor series. The CMSIS-RTOS API is a generic RTOS interface for Cortex-M processor-based devices. You will find details of this standard as part of the Keil installation at Keil\ARM\CMSIS\Documentation\RTOS\html. CMSIS-RTOS provides a standardized API for software components that require RTOS functionality and gives therefore serious benefits to the users and the software industry.

- CMSIS-RTOS provides basic features that are required in many applications



or technologies such as UML or Java (JVM).

- The unified feature set of the CMSIS-RTOS API simplifies sharing of software components and reduces learning efforts.
- Middleware components that use the CMSIS-RTOS API are RTOS agnostic. CMSIS-RTOS compliant middleware is easier to adapt.
- Standard project templates (such as motor control) of the CMSIS-RTOS API may be shipped with freely available CMSIS-RTOS implementations.

## 5.5.4. FreeRTOS

FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller. FreeRTOS only provides the core real-time scheduling functionality, inter-task communication, timing and synchronization primitives. This means it is more accurately described as a real-time kernel, or real-time executive. FreeRTOS is available for 35 processor architectures, with millions of product deployments. For more information on FreeRTOS, see their web site at <http://www.freertos.org/RTOS-Cortex-M3-M4.html>. The starter project for the LM3S811 can be easily recompiled to run on any of the Texas Instruments Cortex M microcontrollers.

FreeRTOS is licensed under a modified GPL and can be used in commercial applications under this license without any requirement to expose your proprietary source code. An alternative commercial license option is also available in cases that: You wish to receive direct technical support. You wish to have assistance with your development. You require legal protection or other assurances. Program 5.10 shows the PendSV handler that implements the context switch. Notice that this thread switch does not disable interrupts. Rather, the **ISB** instruction acts as an instruction synchronization barrier. It flushes the pipeline of the processor, so that all instructions following the **ISB** are fetched from cache or memory again, after the **ISB** instruction has been completed. Similar to Micrium  $\mu$ C/OS-II and ARM RTX, the FreeRTOS does run user threads with the process stack pointer (PSP).

```
__asm void xPortPendSVHandler( void ){  
extern uxCriticalNesting;  
extern pxCurrentTCB;  
extern vTaskSwitchContext;  
PRESERVE8  
mrs r0, psp  
isb  
ldr r3, =pxCurrentTCB /* Get the location of current TCB. */  
ldr r2, [r3]  
stmdb r0!, {r4-r11} /* Save the remaining registers. */  
str r0, [r2] /* Save the new top of stack into the TCB. */
```

```

stmdb sp!, {r3, r14}
mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
msr basepri, r0
bl vTaskSwitchContext
mov r0, #0
msr basepri, r0
ldmia sp!, {r3, r14}
ldr r1, [r3]
ldr r0, [r1] /* first item in pxCurrentTCB is task top of stack. */
ldmia r0!, {r4-r11} /* Pop registers and critical nesting count. */
msr psp, r0
isb
bx r14
nop
}

```

*Program 5.10. Thread switch code on FreeRTOS also uses PendSV for the Cortex M3.*

## 5.5.5. Other Real Time Operating Systems

Other real time operating systems available for the Cortex M are listed in Table 5.3

Provider	Product
<a href="#">CMX Systems</a>	CMX-RTX,CMX-Tiny
Expresslogic	ThreadX
<a href="#">Green Hills</a>	Integrity®, µVelOSity
<a href="#">Mentor Graphics</a>	Nucleus+®
<a href="#">Micro Digital</a>	SMX®
RoweBots	Unison
SEGGER	embOS

**Table 5.3 Other RTOS for the Cortex M (<http://www.ti.com/lsds/ti/tools-software/rtos.page#arm>)**

Deployed in over 1.5 billion devices, VxWorks® by Wind River® is the world's leading real-time operating system (RTOS). It is listed here in the other category because it is deployed on such architectures as the X86, ARM Cortex-A series, and Freescale QorIQ, but not on the Cortex M microcontrollers like the TM4C123. VxWorks delivers hard real-time performance, determinism, and low latency along with the scalability, security, and safety required for aerospace and defense, industrial, medical, automotive, consumer electronics, networking, and other industries. VxWorks has become the RTOS of choice when certification is required. VxWorks supports the space, time, and resource partitioning required for IEC 62304, IEC 61508, IEC 50128, DO-178C, and ARINC 653 certification. VxWorks

customers can design their systems to the required level of security by picking from a comprehensive set of VxWorks security features. VxWorks is an important play in providing solutions for the Internet of Things (IoT), where connectivity, scalability, and security are required. For more information, see <http://www.windriver.com/products/vxworks/>

---

## 5.6. Exercises

5.1 For each of the following terms give a definition in 16 words or less

a) aging	d) least slack time first	g) rate monotonic
b) certification	e) exponential queue	h) Kahn Process
c) starvation	f) maximum latency	Network
		i) monitor

5.2 Select the best term from the book that describes each definition.

- a) A technique to periodically increase the priority of low-priority threads so that low priority threads occasionally get run. The increase is temporary.
- b) A situation that can occur in a priority thread scheduler where a low-priority thread never runs.
- c) The condition where thread 1 is waiting for a unique resource held by thread 2, and thread 2 is waiting for a unique resource held by thread 1.
- d) The condition where a thread is not allowed to run because it needs something that is unavailable.
- e) The condition where once a thread blocks, there are a finite number of threads that will be allowed to proceed before this thread is allowed to proceed.
- f) An operation that once started will run to completion without interruption
- g) An implementation using a FIFO or mailbox that separates data input from data processing.
- h) A technique that could be used to prevent the user from executing I/O on a driver until after the user calls the appropriate initialization.
- i) A scheduling algorithm that assigns priority linearly related to how often a thread needs to run. Threads needing to run more often have a higher priority.
- j) An OS feature that allows the user to run user-defined software at specific places within the OS. These programs are extra for the user's convenience and not required by the OS itself.
- k) An OS feature that allows you to use the OS in safety-critical applications.
- l) A scheduling algorithm with round robin order but varying time slice. If a thread blocks on I/O, its time slice is reduced. If it runs to completion of a time slice, its time slice is increased.
- m) The condition where at most one thread is allowed access to a resource that cannot be shared. If a second thread wishes access to the resource while the first thread is using it, the second thread is made to wait until the first thread is finished.
- n) The condition a function has that allows it to be simultaneously executed by multiple threads.
- o) A thread scheduling algorithm that has the threads themselves decide when the thread switches should occur.
- p) A situation that can occur in a priority thread scheduler where a high-priority

thread is waiting on a resource owned by a low-priority thread.

q) A type of semaphore implemented with a busy-wait loop.

r) A type of thread scheduler where each thread has equal priority and all threads are executed in a circular sequence.

5.3 In this problem you will extend the preemptive scheduler to support priority. This system should support three levels of priority "1" will be the highest. You can solve this problem using either assembly or C.

a) Redesign the TCB to include a 32-bit integer for the priority (although the values will be restricted to 1,2,3). Show the static allocation for the three threads from the example in this chapter assuming the first two are priority 2 and the last is priority 3. There are no priority 1 threads in this example, but there might be in the future.

b) Redesign the scheduler to support this priority scheme.

c) In the book it said "*Normally, we add priority to a system that implements blocking semaphores and not to one that uses spinlock semaphores.*" What specifically will happen here if the system is run with spinlock semaphores?

d) Even when the system supports blocking semaphores, starvation might happen to the low priority threads. Describe the sequence of events that cause starvation.

e) Suggest a solution to the starvation problem.

5.4 This problem investigates the design of an adaptive priority scheduler with exponential time slices. This is also called an *exponential Queue* or *multi-level feedback queue*. The CTSS system (MIT, early 1960's) was the first to use exponential queues. One of the difficulties in a priority scheduler is the assignment of priority. Typically, one wishes to assign a high priority to threads doing I/O (which block a lot) so that the response to I/O is short, and assign a low priority to threads not doing I/O (which do not block a lot). However, in a complex system a particular thread may sometimes exhibit I/O bound behavior, but later exhibit CPU bound behavior. An adaptive scheduler will adjust the priority according to the current activity of the thread. Priority 1 threads will run with a time slice of 4000 (1ms), priority 2 threads will run with a time slice of 8000 (2ms), and priority 3 threads will run with a time slice of 16000 (4ms). Consider this blocking round-robin scheduler, with two new entries, shown in **bold**, added to the TCB.

```
struct TCB{
    struct TCB *Next; // Link to Next TCB
    int32_t *StackPt; // Stack Pointer
    Sema4Type *BlockPt; // 0 if not blocked, pointer if blocked
    int16_t Priority; // 1 (highest), 2, or 3 (lowest)
    uint16_t TimeSlice; // 4000,8000, or 16000
    int32_t Stack[100]; // stack, size determined at runtime
};
typedef struct TCB TCBType;
typedef TCBType * TCBPtr;
```

a) Rewrite the **OS\_Wait** function so that if a priority 2 or 3 thread blocks, its priority

will be raised (decrement by 1) and its time slice will be halved. No changes to **OS\_Signal** will be needed.

**b)** Rewrite the **threadSwitch** ISR so that if a priority 1 or 2 thread runs to the end of its time slice without blocking, its priority will be lowered (increment by 1) and its time slice will be doubled. In addition, implement priority scheduling with variable time slices.

5.5 Consider the implementation of **OS\_AddThreads**, shown in Program 3.4. Redesign the system so that if the user program finishes, the OS will run the user program again. For example, this user function executes **stuff1**, **stuff2** and **stuff3** once and quits.

```
void user(void){ stuff1(); stuff2(); stuff3();}
```

If the user calls this system function to activate **user**,

```
OS_AddThreads(&user);
```

then with your updated system **stuff1**, **stuff2** and **stuff3** will be repeated over and over again. You are allowed to make changes to the **struct** and to **OS\_AddThreads**, but not to **user** or other OS functions. You can however add additional OS functions. In particular, show changes to the **struct** and rewrite **OS\_AddThreads** in its entirety.

# 6. Digital Signal Processing

## Chapter 6 objectives are to:

- Introduce basic principles involved in digital filtering
- Define the Z Transform and use it to design and analyze digital filters
- Present the discrete Fourier Transform and use it to design digital filters
- Develop digital filter implementations
- Present an audio input/output example

The goal of this chapter is to provide a brief introduction to digital signal processing (DSP). DSP includes a wide range of operations such as digital filtering, event detection, frequency spectrum analysis and signal compression/decompression. Similar to the goal of analog filtering, a digital filter will be used to improve the signal to noise ratio in our data. The difference is that a digital filter is performed in software on the digital data sampled by the ADC converter. The particular problem addressed in a couple of ways in this chapter is removing 60 Hz noise from the signal. Like the control systems and communication systems discussed elsewhere in these volumes, we will provide just a brief discussion to the richly developed discipline of DSP. Again, this chapter focuses mostly on the implementation on the embedded microcomputer. Event detection is the process of identifying the presence or absence of particular patterns in our data. Examples of this type of processing include optical character readers, waveform classification, sonar echo detection, infant apnea monitors, heart arrhythmia detectors and burglar alarms. Frequency spectrum analysis requires the calculation of the Discrete Fourier Transform (DFT). A fast algorithm to calculate the DFT is called the Fast Fourier Transform, FFT. Like the regular Fourier Transform, the DFT converts a time-dependent signal into the frequency domain. The difference a regular Fourier Transform and the DFT is that the DFT performs the conversion on a finite number of discrete time digital samples to give a finite number of points at discrete frequencies. We will use the DFT in this chapter as a flexible way to design digital filters. Data compression and decompression are important aspects in high-speed communication systems. Although we will not specifically address the problems of event detection, DFT and compression/decompression in this book, these DSP operations are implemented using similar techniques as the digital filters that are presented in this chapter. The goal of this chapter is to demonstrate that fairly powerful digital signal processing techniques can be implemented on most microcontrollers.

---

## 6.1. Basic Principles

The objective of this section is to introduce simple digital filters. Let  $x_c(t)$  be the continuous analog signal to be digitized.  $x_c(t)$  is the analog input to the ADC converter. If  $f_s$  is the sample rate, then the computer samples the ADC every  $T$  seconds. ( $T = 1/f_s$ ). Let  $\dots, x(n), \dots$  be the ADC output sequence, where

$$x(n) = x_c(nT) \quad \text{with } -\infty < n < +\infty.$$

There are two types of approximations associated with the sampling process. Because of the finite precision of the ADC, amplitude errors occur when the continuous signal,  $x_c(t)$ , is sampled to obtain the digital sequence,  $x(n)$ . The second type of error occurs because of the finite sampling frequency. The Nyquist Theorem states that the digital sequence,  $x(n)$ , properly represents the DC to  $\frac{1}{2}f_s$  frequency components of the original signal,  $x_c(t)$ . There are two important assumptions that are necessary to make when using digital signal processing:

1. We assume the signal has been sampled at a fixed and known rate,  $f_s$ .
2. We assume aliasing has not occurred.

We can guarantee the first assumption by using a hardware clock to start the ADC at a fixed and known rate. A less expensive but not as reliable method is to implement the sampling routine as a high priority periodic interrupt process. If the time jitter is  $\delta t$  then we can estimate the voltage error by multiplying the time jitter by the slew rate of the input,  $\partial V/\partial t * \delta t$ . By establishing a high priority of the interrupt handler, we can place an upper bound on the interrupt latency, guaranteeing that ADC sampling is occurring at an almost fixed and known rate. We can observe the ADC input with a spectrum analyzer to prove there are no significant signal components above  $\frac{1}{2}f_s$ . “No significant signal components” is defined as having an ADC input voltage  $|Z|$  less than the ADC resolution,  $\Delta z$ ,

$$|Z| \leq \Delta z \quad \text{for all } f \geq \frac{1}{2}f_s$$

A **causal** digital filter calculates  $y(n)$  from  $y(n-1), y(n-2), \dots$  and  $x(n), x(n-1), x(n-2), \dots$ . Simply put, a causal filter cannot have a nonzero output until it is given a nonzero input. The output of a causal filter,  $y(n)$ , cannot depend on future data (e.g.,  $y(n+1), x(n+1)$  etc.)

A **linear** filter is constructed from a linear equation. A **nonlinear** filter is constructed from a nonlinear equation. An example of a nonlinear filter is the median. To



calculate the median of three numbers, one first sorts the numbers according to magnitude, then chooses the middle value. Other simple nonlinear filters include maximum, minimum, and square.

A **finite impulse response** filter (FIR) relates  $y(n)$  only in terms of  $x(n)$ ,  $x(n-1)$ ,  $x(n-2)$ ,... If the sampling rate is 360 Hz, this simple FIR filter will remove 60 Hz noise:

$$y(n) = (x(n) + x(n-3))/2$$

An **infinite impulse response** filter (IIR) relates  $y(n)$  in terms of both  $x(n)$ ,  $x(n-1)$ ,..., and  $y(n-1)$ ,  $y(n-2)$ ,... This simple IIR filter has averaging or low-pass behavior:

$$y(n) = (x(n) + y(n-1))/2$$

One way to analyze linear filters is the Z-Transform. The definition of the Z-Transform is:

$$X(z) = Z[x(n)] \equiv \sum_{n=-\infty}^{+\infty} (x(n) * z^{-n})$$

The Z-transform is similar to other transforms. In particular, consider the Laplace Transform, which converts a continuous time-domain signal,  $x(t)$ , into the frequency domain,  $X(s)$ . In the same manner, the Z-Transform converts a discrete time sequence,  $x(n)$ , into the frequency domain,  $X(z)$ . See Figure 6.1.

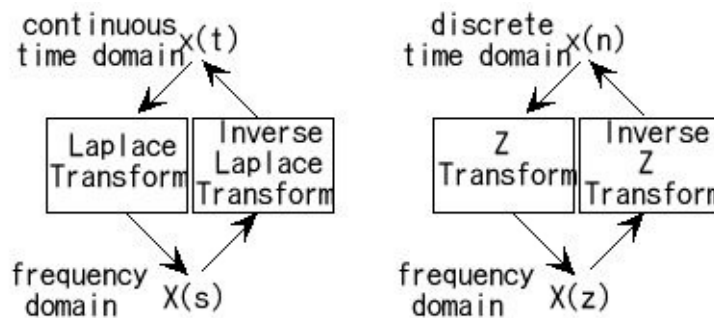


Figure 6.1. A transform is used to study a signal in the frequency domain.

The input to both the Laplace and Z Transforms are infinite time signals, having values at times from  $-\infty$  to  $+\infty$ . The frequency parameters,  $s$  and  $z$ , are complex numbers, having real and imaginary parts. In both cases we apply the transform to study linear systems. In particular, we can describe the behavior (gain and phase) of an analog system using its transform,  $H(s) = Y(s)/X(s)$ . In this same way we will use the  $H(z)$  transform of a digital filter to determine its gain and phase response. See Figure 6.2.

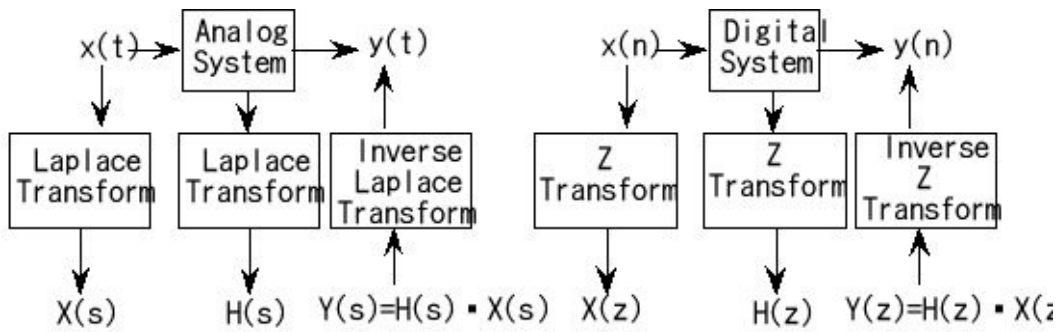


Figure 6.2. A transform can also be used to study a system in the frequency domain.

For an analog system we can calculate the gain by taking the magnitude of  $H(s)$  at  $s = j 2\pi f$ , for all frequencies,  $f$ , from  $-\infty$  to  $+\infty$ . The phase will be the angle of  $H(s)$  at  $s = j 2\pi f$ . If we were to plot the  $H(s)$  in the  $s$  plane, the  $s = j 2\pi f$  curve is the entire  $y$ -axis. For a digital system we will calculate the gain and phase by taking the magnitude and angle of  $H(z)$ . Because of the finite sampling interval, we will only be able to study frequencies from DC to  $\frac{1}{2}f_s$  in our digital systems. If we were to plot the  $H(z)$  in the  $z$  plane, the  $z$  curve representing the DC to  $\frac{1}{2}f_s$  frequencies will be the unit circle,  $z \equiv e^{j2\pi f/f_s}$ .

We will begin by developing a simple, yet powerful rule that will allow us to derive the  $H(z)$  transforms of most digital filters. Let  $m$  be an integer constant. We can use the definition of the Z-Transform to prove that:

$$\begin{aligned}
 Z[x(n-m)] &= \sum(x(n-m) * z^{-n}) \quad \text{for } n = -\infty \text{ to } +\infty \\
 &= \sum(x(p) * z^{-p-m}) \quad \text{let } p = n-m, n = p+m \\
 &= z^{-m} * \sum(x(p) * z^{-p}) \quad \text{because } m \text{ is a constant} \\
 &= z^{-m} Z[x(n)] \quad \text{by definition of Z-Transform}
 \end{aligned}$$

For example, if  $X(z)$  is the Z-Transform of  $x(n)$ , then  $z^{-3} \cdot X(z)$  is the Z-Transform of  $x(n-3)$ . To find the Z-Transform of a digital filter, take the transform of both sides of the linear equation and solve for

$$H(z) \equiv Y(z) / X(z)$$

To find the response of the filter, let  $z$  be a complex number on the unit circle

$$z = e^{j2\pi f/f_s} = \cos(2\pi f/f_s) + j \sin(2\pi f/f_s) \quad \text{for } 0 \leq f < \frac{1}{2}f_s$$

Let  $H(f) = a + bj$ , where  $a$  and  $b$  are real numbers. The gain of the filter is the complex magnitude of  $H(z)$  as  $f$  varies from 0 to  $\frac{1}{2}f_s$ .

$$\text{Gain} \equiv |H(f)| = \text{sqrt}(a^2 + b^2)$$

The **phase response** of the filter is the angle of  $H(z)$  as  $f$  varies from 0 to  $\frac{1}{2}f_s$ .

$$\text{Phase} \equiv \text{angle}[H(f)] = \tan^{-1}(b/a)$$

Another way to analyze digital filters is to consider the filter response to particular input sequences. Two typical sequences are the step and the impulse (Figure 6.3).

step           ..., 0, 0, 0, 1, 1, 1, 1, ...  
 impulse       ..., 0, 0, 0, 1, 0, 0, 0, ...

The **impulse** is defined as:

$$\begin{aligned} i(n) &\equiv 1 && \text{for } n = 0 \\ &0 && \text{for } n \neq 0 \end{aligned}$$

The **step** is defined as:

$$\begin{aligned} s(n) &\equiv 0 && \text{for } n < 0 \\ &1 && \text{for } n \geq 0 \end{aligned}$$

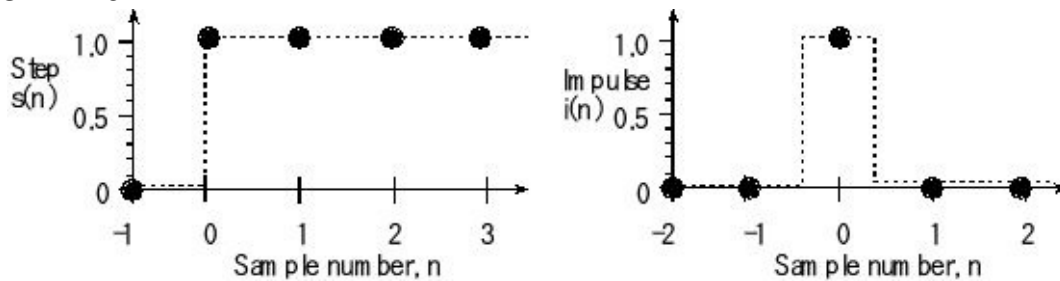


Figure 6.3. Step and impulse inputs.

The step signal represents a sharp change (like an edge in a photograph). We will analyze three digital filters. The FIR is  $y(n) = (x(n)+x(n-1))/2$ . The IIR is  $y(n) = (x(n)+y(n-1))/2$ . The nonlinear filter is  $y(n) = \text{median}(x(n), x(n-1), x(n-2))$ . The median can be performed on any odd number of data points by sorting the data and selecting the middle value. The median filter can be performed recursively or nonrecursively. A nonrecursive 3-wide median filter is implemented in Program 6.1.

```
uint8_t Median(uint8_t u1,uint8_t u2,uint8_t u3){
uint8_t result;
if(u1>u2)
if(u2>u3) result = u2; // u1>u2,u2>u3    u1>u2>u3
else
if(u1>u3) result = u3; // u1>u2,u3>u2,u1>u3 u1>u3>u2
else result = u1; // u1>u2,u3>u2,u3>u1 u3>u1>u2
else
if(u3>u2) result = u2; // u2>u1,u3>u2    u3>u2>u1
else
if(u1>u3) result = u1; // u2>u1,u2>u3,u1>u3 u2>u1>u3
else result = u3; // u2>u1,u2>u3,u3>u1 u2>u3>u1
return(result);
}
```

Program 6.1: The median filter is an example of a nonlinear filter.

For a nonrecursive median filter, the original data points are not modified. For example, a 5-wide nonrecursive median filter takes as the filter output the median of  $\{x(n), x(n-1), x(n-2), x(n-3), x(n-4)\}$ . On the other hand, a recursive median filter replaces the sample point with the filter output. For example, a 5-wide recursive median filter takes as the filter output the median of  $\{x(n), y(n-1), y(n-2), y(n-3), y(n-4)\}$  where  $y(n-1), y(n-2), \dots$  are the previous filter outputs. A median filter can be applied in systems that have impulse or speckle noise. For example, the noise every once in a while causes one sample to be very different than the rest (like a speck on a piece of paper) then the median filter will completely eliminate the noise. Except for the delay, the median filter passes a step without error. The step responses of the three filters are (Figure 6.4):

FIR            ..., 0, 0, 0, 0.5, 1, 1, 1, ...  
 IIR            ..., 0, 0, 0, 0.5, 0.75, 0.88, 0.94, 0.97, 0.98, 0.99, ...  
 median        ..., 0, 0, 0, 0, 1, 1, 1, 1, 1, ...

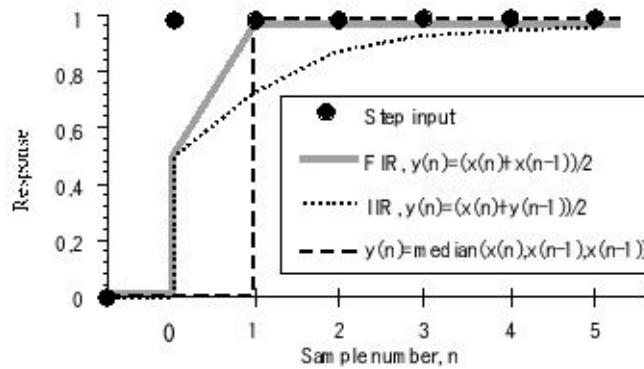


Figure 6.4. Step response of three simple digital filters.

The impulse represents a noise spike (like spots on a Xerox copy). The impulse response of a filter is defined as  $h(n)$ . The median filter completely removes the impulse. The impulse responses of the three filters are (Figure 6.5):

FIR            ..., 0, 0, 0, 0.5, 0.5, 0, 0, 0, ...  
 IIR            ..., 0, 0, 0, 0.5, 0.25, 0.13, 0.06, 0.03, 0.02, 0.01, ...  
 median        ..., 0, 0, 0, 0, 0, 0, 0, 0, ...

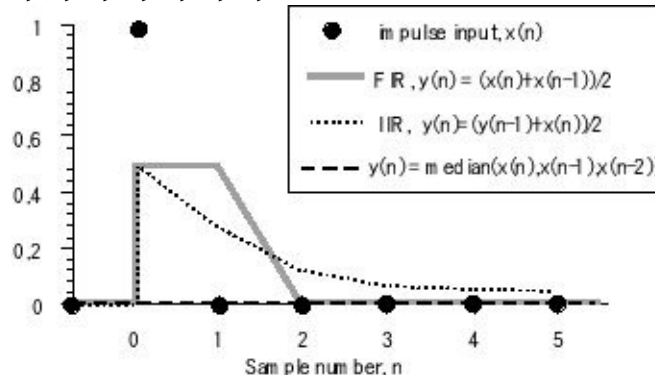


Figure 6.5. Impulse response of three simple digital filters.

Note that the median filter preserves the sharp edges and removes the spike or impulsive noise. The median filter is **nonlinear**, and hence  $H(z)$  and  $h(n)$  are not

defined for this particular class of filters. For linear filters, the impulse response,  $h(n)$ , can also be used as an alternative to the transfer function  $H(z)$ .  $h(n)$  is sometimes called the **direct form**. A causal filter has  $h(n) = 0$  for  $n$  less than 0. For a casual filter.

$$H(z) = \sum (h(n) * z^{-n}) \text{ for } n=0 \text{ to } +\infty$$

For a finite impulse response (FIR) filter,  $h(n) = 0$  for  $n \geq N$  for some finite  $N$ . Thus,

$$H(z) = \sum (h(n) * z^{-n}) \text{ for } n=0 \text{ to } N-1$$

The output of a filter can be calculated by convolving the input sequence,  $x(n)$ , with  $h(n)$ . For an infinite impulse response filter:

$$y(n) = \sum (h(n) * x(n-k)) \text{ for } n=0 \text{ to } +\infty$$

For a finite impulse response (FIR) filter:

$$y(n) = \sum (h(n) * x(n-k)) \text{ for } n=0 \text{ to } N-1$$

## 6.2. Multiple Access Circular Queue

A multiple access circular queue (**MACQ**) is used for data acquisition and control systems. A MACQ is a fixed length order preserving data structure, see Figure 6.6. The source process (ADC sampling software) places information into the MACQ. Once initialized, the MACQ is always full. The oldest data is discarded when the newest data is **Put** into a MACQ. The sink process can read any of the data from the MACQ. The **Read** function is non-destructive. This means that the MACQ is not changed by the **Read** operation. In this MACQ, the newest sample,  $x(n)$ , is stored in element  $\mathbf{x}[0]$ .  $x(n-1)$ , is stored in element  $\mathbf{x}[1]$ .

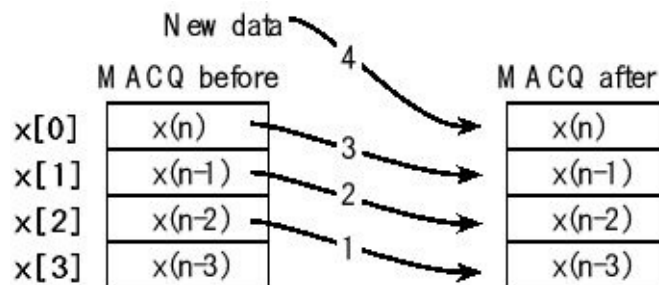


Figure 6.6. When data is put into a multiple access circular queue, the oldest data is lost.

To **Put** data into this MACQ, four steps are followed, as shown in Figure 6.6. First, the data is shifted down (steps 1, 2, 3), and then the new data is entered into the  $\mathbf{x}[0]$  position (step 4).

The drawing in Figure 6.6 shows the position in memory of  $x(n)$ ,  $x(n-1)$ ,... does not move when one data sample is added. Notice however, the data itself does move. As time passes the data gets older, the data moves down in the MACQ.

A simple application of the MACQ is the real-time calculation of derivative. Also assume the ADC sampling is triggered every 1 ms.  $x(n)$  will refer to the current sample, and  $x(n-1)$  will be the sample 1 ms ago. There are a couple of ways to implement a discrete time derivative. The simple approach is

$$d(n) = (x(n) - x(n-1)) / \Delta t$$

In practice, this first order equation is quite susceptible to noise. An approach generating less noise calculates the derivative using a higher order equation like

$$d(n) = (x(n) + 3x(n-1) - 3x(n-2) - x(n-3)) / \Delta t$$

The C implementation of this discrete derivative uses a MACQ (Program 6.2). Since  $\Delta t$  is 1 ms, we simply consider the derivative to have units mV/ms and not actually execute the divide by  $\Delta t$  operation. Signed arithmetic is used because the slope may be negative.

```

int32_t x[4]; // MACQ (mV)
int32_t d; // derivative(V/s)
void ADC3_Handler(void){
    ADC_ISC_R = 0x08; // acknowledge ADC sequence 3 completion
    x[3] = x[2]; // shift data
    x[2] = x[1]; // units of mV
    x[1] = x[0];
    x[0] = (3000*ADC_SSFIFO3_R)>>12; // in mV
    d = x[0]+3*x[1]-3*x[2]-x[3]; // in V/s
    Fifo_Put(d); // pass to foreground
}

```

*Program 6.2. Software implementation of first derivative using a multiple access circular queue.*

When the MACQ holds many data points, it can be implemented using a pointer or index to the newest data. In this way, the data need not be shifted each time a new sample is added. The disadvantage of this approach is that address calculation is required during the **Read** access. For example, we could implement a 16-element averaging filter. More specifically, we will calculate the average of the last 16 samples, see Program 6.3.

Entering data into this MACQ is a three step process (Figure 6.7). First, the pointer is decremented. If necessary, the pointer is wrapped such that it is always pointing to elements **x[0]** through **x[15]** . Second, new data is stored into the location of the pointer. Third, a second copy of the new data is stored 16 elements down from the pointer.

Because the pointer is maintained within the first 16 elements, **\*Pt** to **\*(Pt+15)** will always point to valid data within the MACQ. Let  $m$  be an integer from 0 to 15. In this MACQ, the data element  $x(n-m)$  can be found using **\*(Pt+m)** .

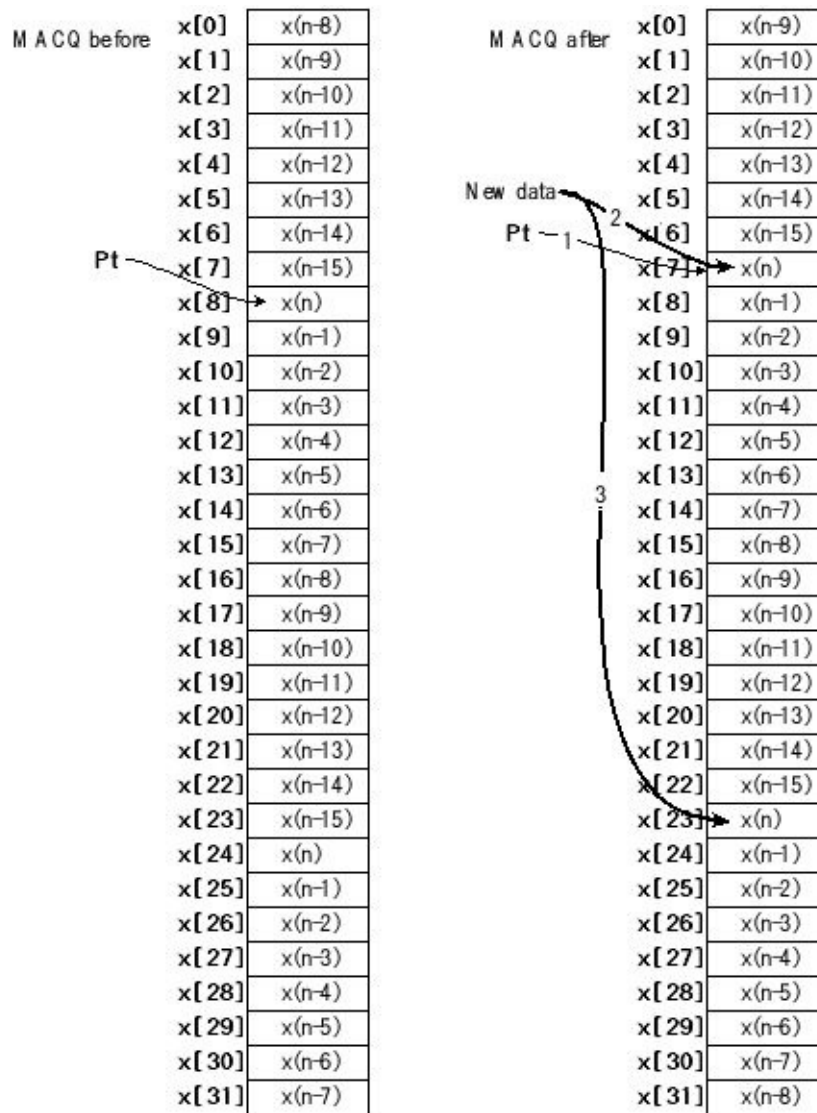


Figure 6.7. When data is put into a multiple access circular queue, the oldest data is lost.

Figure 6.7 shows the labels  $x(n)$ ,  $x(n-1)$ ,... moving from before to after. Notice however, the data itself does not move. What moves is the significance (or meaning) of the data. The data grows older as time passes. The passage of time is produced by decrementing the pointer. Having two copies of the data makes reading the data faster, because the operation  $*(Pt+m)$  never needs wrapping.

**Observation:** It is possible to implement a pointer-based MACQ that keeps just one copy of the data. Time to access data would be slower, but half as much storage would be needed.

```

uint16_t x[32]; // two copies
uint16_t *Pt; // pointer to current
uint16_t Sum; // sum of the last 16 samples
void LPF_Init(void){
    Pt = &x[0]; Sum = 0;
}

```



```

// calculate one filter output, called at sampling rate
// Input: new ADC data  Output: filter output, DAC data
uint16_t LPF_Calc(uint16_t newdata){
    Sum = Sum - *(Pt+16); // subtract the one 16 samples ago
    if(Pt == &x[0]){
        Pt = &x[16]; // wrap
    } else{
        Pt--; // make room for data
    }
    *Pt = *(Pt+16) = newdata; // two copies of the new data
    return Sum/16;
}

```

*Program 6.3. Digital low pass filter implemented by averaging the previous 16 samples (cutoff =  $f_s/32$ ).*

## 6.3. Using the Z-Transform to Derive Filter Response

In this section, we will use the Z-Transform to determine the digital filter response (gain and phase) given the filter equation. The first example is the average of the current sample with the sample 3 times ago. Program 6.4 shows the implementation.

$$y(n) = (x(n) + x(n-3))/2$$

The first step is to take the Z-Transform of both sides of the equation. The Z-Transform of  $y(n)$  is  $Y(z)$ , the Z-Transform of  $x(n)$  is  $X(z)$ , and the Z-Transform of  $x(n-3)$  is  $z^{-3}X(z)$ . Since the Z-Transform is a linear operator, we can write:

$$Y(z) = (X(z) + z^{-3}X(z))/2$$

The next step is to rewrite the equation in the form of  $H(z) \equiv Y(z)/X(z)$ .

$$H(z) \equiv Y(z)/X(z) = 1/2 (1 + z^{-3})$$

We plug in  $z \equiv e^{j2\pi f/f_s}$  calculate the gain and phase response, see Figures 6.8 and 6.9.

$$H(f) = 1/2 (1 + e^{-j6\pi f/f_s}) = 1/2 (1 + \cos(6\pi f/f_s) - j \sin(6\pi f/f_s))$$

$$\text{Gain} \equiv |H(f)| = 1/2 \sqrt{(1 + \cos(6\pi f/f_s))^2 + \sin(6\pi f/f_s)^2}$$

$$\text{Phase} \equiv \text{angle}(H(f)) = \tan^{-1}(-\sin(6\pi f/f_s)/(1 + \cos(6\pi f/f_s)))$$

```
int32_t x[4]; // MACQ
void ADC3_Handler(void){ int32_t y;
    ADC_ISC_R = 0x08; // acknowledge ADC sequence 3 completion
    x[3] = x[2]; // shift data
    x[2] = x[1]; // units, ADC sample 0 to 4095
    x[1] = x[0]; // see chapter 1 for details on the ADC
    x[0] = ADC_SSFIFO3_R; // 0 to 4095
    y = (x[0]+x[3])/2; // filter output
    Fifo_Put(y); // pass to foreground
}
```

*Program 6.4. If the sampling rate is 360 Hz, this filter rejects 60 Hz.*

**Checkpoint 6.1:** If the sampling rate in Program 6.4 is 360 Hz, use the Z transform to prove the 60 Hz gain is zero.

**Observation:** Program 6.4 is double notch filter rejecting  $1/6$  and  $1/2 f_s$ .

The second example is the average of the current sample with the previous filter output. Program 6.5 shows the implementation

$$y(n) = (x(n) + y(n-1))/2$$

The first step is to take the Z-Transform of both sides of the equation. The Z-Transform of  $y(n)$  is  $Y(z)$ , the Z-Transform of  $x(n)$  is  $X(z)$ , and the Z-Transform of  $y(n-1)$  is  $z^{-1}Y(z)$ . Since the Z-Transform is a linear operator, we can write:

$$Y(z) = (X(z) + z^{-1}Y(z))/2$$

The next step is to rewrite the equation in the form of  $H(z) \equiv Y(z)/X(z)$ .

$$H(z) \equiv Y(z)/X(z) = 1/(2 - z^{-1})$$

We plug in  $z \equiv e^{j2\pi f/f_s}$  calculate the gain and phase response, see Figures 6.8 and 6.9.

$$H(f) = 1/(2 - e^{-j2\pi f/f_s}) = 1/(2 - \cos(2\pi f/f_s) + j \sin(2\pi f/f_s))$$

$$\text{Gain} \equiv |H(f)|$$

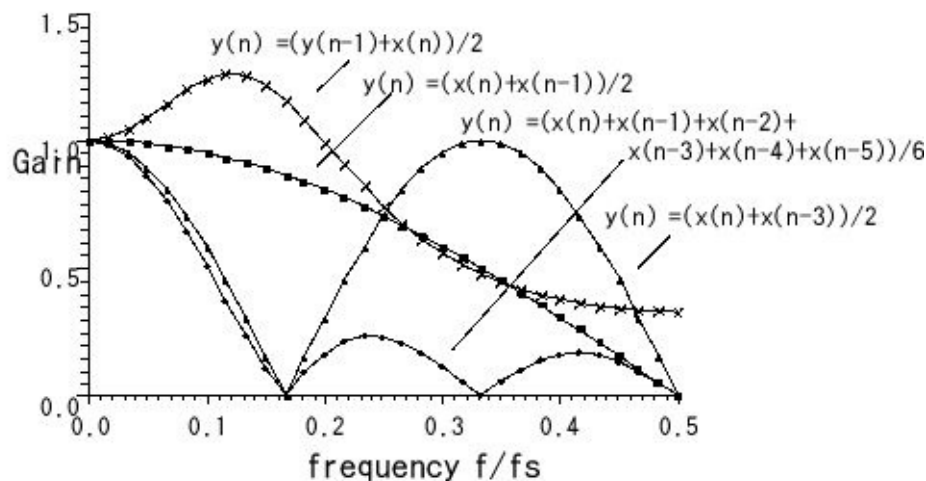
$$\text{Phase} \equiv \text{angle}(H(f))$$

```
int32_t y;
void ADC3_Handler(void){ int32_t x;
  ADC_ISC_R = 0x08; // acknowledge ADC sequence 3 completion
  x = ADC_SSFIFO3_R; // 0 to 4095
  y = (x+y)/2; // filter output
  Fifo_Put(y); // pass to foreground
}
```

*Program 6.5. Implementation of an IIR low pass digital filter.*

**Checkpoint 6.2:** For  $f$  between 0 and  $0.2 f_s$ , the filter in Program 6.5 has a gain larger than 1 (see Figure 6.8). What does that mean?

The gain of four linear digital filters is plotted in Figure 6.8 and the phase response is plotted in Figure 6.9.



*Figure 6.8. Gain versus frequency response for four simple digital filters.*

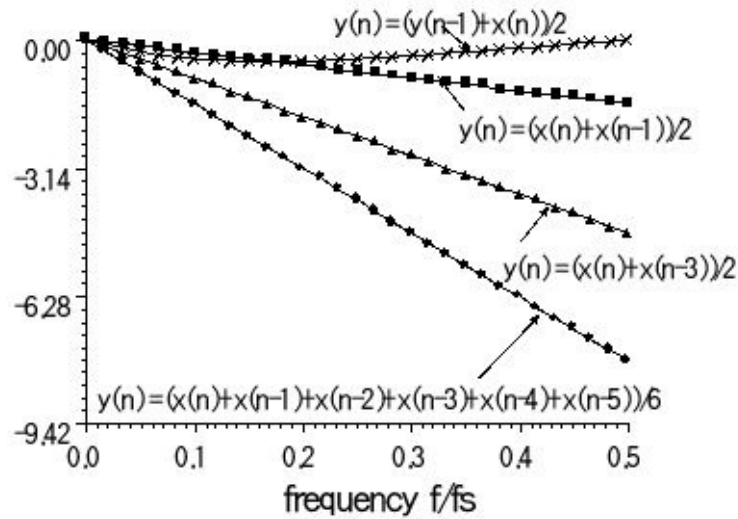


Figure 6.9. Phase versus frequency response for four simple digital filters.

A linear phase versus frequency response is desirable because a linear phase causes minimal waveform distortion. Conversely, a nonlinear phase response will distort shape or morphology of the signal. In general, if  $f_s$  is  $2 \cdot k \cdot f_c$  Hz (where  $k$  is any integer  $k \geq 2$ ), then the following is a  $f_c$  notch filter:

$$y(n) = (x(n) + x(n-k)) / 2$$

Averaging the last  $k$  samples will perform a low-pass filter with notches. Let  $f_c$  be the frequency we wish to reject. We will choose the sampling at a multiple of this notch. I.e., we choose  $f_s$  to be  $k \cdot f_c$  Hz (where  $k$  is any integer  $k \geq 2$ ), then the  $k$ -sample average filter will reject  $f_c$  and its harmonics:  $2f_c, 3f_c, \dots$ . If the number of terms  $k$  is large, the straight forward implementation of average will run slowly. Fortunately, this averaging filter can be rewritten as a function of the current sample  $x(n)$ , the sample  $k$  times ago  $x(n-k)$ , and the last filter output  $y(n-1)$ . This filter with  $k=16$  was implemented in Program 6.3.

$$y(n) = (1/k) * \text{sum}(x(n-i)) \text{ for } i = 0 \text{ to } k-1$$

$$= (x(n) - x(n-k)) / k + y(n-1)$$

The second formulation looks like an IIR filter, but it is a FIR filter because the equations are identical. The  $H(z)$  transfer function for this  $k$ -term averaging filter is

$$H(z) = (1/k) * (1 - z^{-k}) / (1 - z^{-1})$$

This class of digital low-pass filters can be implemented with a  $k+1$  multiple access circular queue, and a simple calculation. The gain of this class of filter is shown in Figure 6.10 for a sampling rate of 100 Hz.

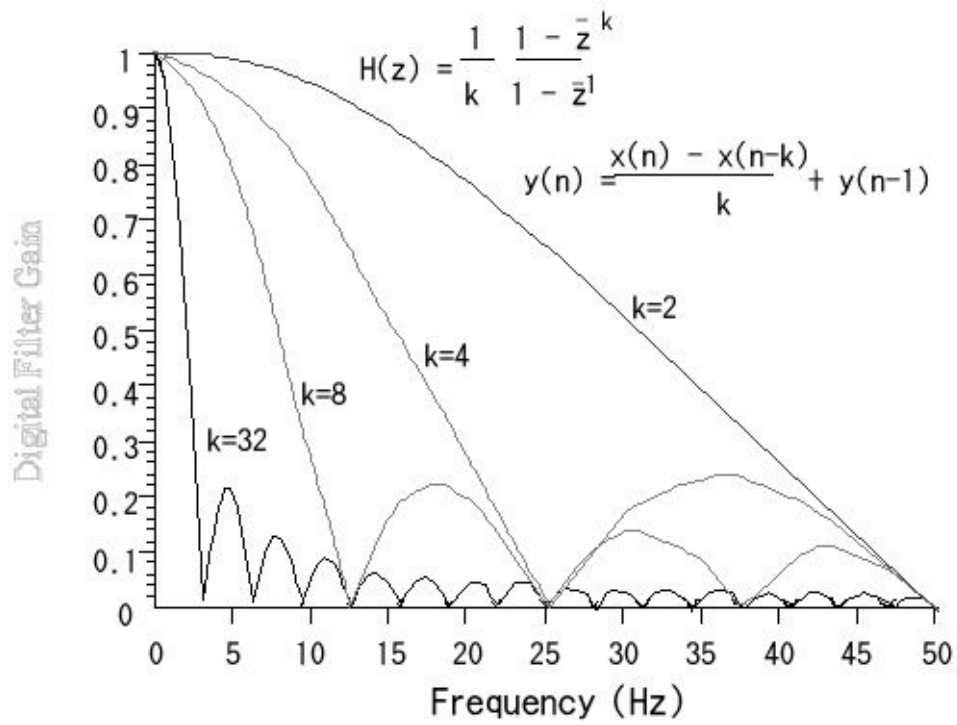


Figure 6.10. Gain versus frequency plot of four averaging low-pass filters.

## 6.4. IIR Filter Design Using the Pole-Zero Plot

The objective of this section is to show the IIR filter design method using pole-zero plots. One starts with a basic shape in mind, and places poles and zeros to generate the desired filter. Consider again the analogy between the Laplace and Z Transforms. When the  $H(s)$  transform is plotted in the  $s$  plane, we look for peaks (places where the amplitude  $H(s)$  is high) and valleys (places where the amplitude is low.) In particular, we usually can identify zeros ( $H(s)=0$ ) and poles ( $H(s)=\infty$ ). A **zero** is a place where  $H(s)=0$ . A **pole** is a place where  $H(s)=\infty$ . In the same way we can plot the  $H(z)$  in the  $z$  plane and identify the poles and zeros. Table 6.1 lists the filter design strategies.

Analog condition	Digital condition	Consequence
zero near $s=j2\pi f$ line	zero near $z=e^{j2\pi f/fs}$	low gain at the $f$ near the zero
pole near $s=j2\pi f$ line	pole near $z=e^{j2\pi f/fs}$	high gain at the $f$ near the pole
zeros in complex conjugate pairs	zeros in complex conjugate pairs	the output $y(t)$ is real
poles in complex conjugate pairs	poles in complex conjugate pairs	the output $y(t)$ is real
poles in left half plane	poles inside unit circle	stable system
poles in right half plane	poles outside unit circle	unstable system
pole near a zero	pole near a zero	high Q response
pole away from a zero	pole away from a zero	low Q response

**Table 6.1. Analogies between the analog and digital filter design rules.**

Once the poles and zeros are placed, the transform of the filter can be written

$$H(z) = \prod \frac{(z - z_i)}{(z - p_i)}$$

where  $z_i$  are the zeros and  $p_i$  are the poles

The first example of this method will be a digital notch filter. 60 Hz noise is a significant problem in most data acquisition systems. The 60 Hz noise reduction can be accomplished:

- 1) Reducing the noise source, e.g., shut off large motors
- 2) Shielding the transducer, cables, and instrument
- 3) Implement a 60 Hz analog notch filter

#### 4) Implement a 60 Hz digital notch filter

The digital notch filter will be more effective and less expensive than an analog notch filter. The signal is sampled at  $f_s$ . We wish to place the zeros (gain=0) at 60 Hz, thus

$$\theta = \pm 2\pi \cdot 60/f_s$$

The zeros are located on the unit circle at 60 Hz

$$z_1 = \cos(\theta) + j \sin(\theta) \quad z_2 = \cos(\theta) - j \sin(\theta)$$

To implement a flat pass band away from 60 Hz the poles are placed next to the zeros, just inside the unit circle. Let  $\alpha$  define the closeness of the poles where  $0 < \alpha < 1$  (Figure 6.11).

$$p_1 = \alpha z_1 \quad p_2 = \alpha z_2$$

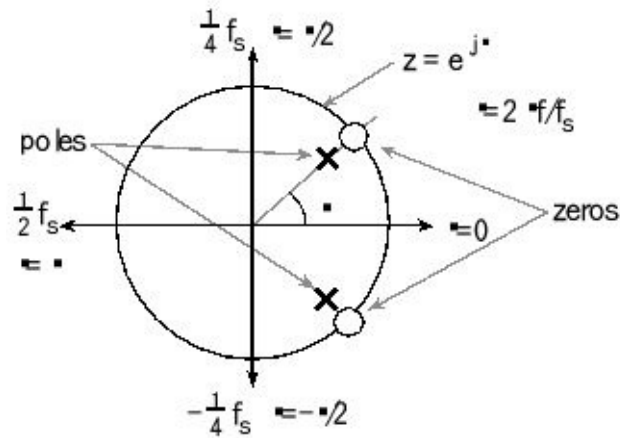


Figure 6.11. Pole-zero plot of a 60 Hz digital notch filter.

The transfer function is

$$H(z) = \prod \frac{(z - z_i)}{(z - p_i)} = \frac{(z - z_1)(z - z_2)}{(z - p_1)(z - p_2)}$$

which can be put in standard form (i.e., with terms 1,  $z^{-1}$ ,  $z^{-2}$  ...)

$$H(z) = \frac{1 - 2\cos(\theta)z^{-1} + z^{-2}}{1 - 2\alpha\cos(\theta)z^{-1} + \alpha^2 z^{-2}}$$

The digital filter can be derived by taking the inverse Z-transform of the  $H(z)$  equation

$$y(n) = x(n) - 2\cos(\theta)x(n-1) + x(n-2) + 2\alpha\cos(\theta)y(n-1) - \alpha^2 y(n-2)$$

Sometimes we can choose  $f_s$  and/or  $\alpha$  to simplify the digital filter equation. For example, if we choose  $f_s = 240$  Hz, then the “ $\cos(\theta)$ ” terms become zero. If we choose  $\alpha = 7/8$  then the fixed-point digital filter becomes:

$$y(n) = x(n) + x(n-2) - (49*y(n-2))/64$$

Another consideration for this type of filter is the fact that the gain in the pass bands is greater than one. The DC gain can be determined two ways. The first method is to use the  $H(z)$  transfer equation and set  $z=1$ . The  $H(z)$  transfer equation for the filter is

$$H(z) = (1+z^{-2})(1 + (49/64)z^{-2})$$

At  $z=1$  this reduces to

$$\text{DC Gain} = (2)(1 + (49/64)) = 128/113$$

The second method to calculate DC gain operates on the filter equation directly. In the first step, we set all  $x(n-k)$  terms in the filter to a single variable “ $x$ ” and all  $y(n-k)$  terms in the filter to a single variable “ $y$ ”. Next we solve for the DC gain, which is  $y/x$ .

$$y = x + x - (49y)/64$$

This method also calculates the DC gain to be 128/113. We can adjust the digital filter so that the DC gain is exactly 1, by prescaling the input terms ( $x(n)$ ,  $x(n-1)$ ,  $x(n-2)$ ,...) by 113/128.

$$y(n) = (113*x(n) + 113*x(n-2) - 98*y(n-2))/128$$

```
int32_t x[3]; // MACQ for the ADC input data
int32_t y[3]; // MACQ for the digital filter output
void ADC3_Handler(void){
    ADC_ISC_R = 0x08; // acknowledge ADC sequence 3 completion
    x[2] = x[1]; x[1] = x[0]; // shift data
    y[2] = y[1]; y[1] = y[0];
    x[0] = ADC_SSFIFO3_R; // 0 to 4095
    y[0] = (113*(x[0]+x[2])-98*y[2])/128; // filter output
    Fifo_Put((int16_t)y[0]); // pass to foreground
}
```

*Program 6.6. If the sampling rate is 240 Hz, this filter rejects 60 Hz.*

Since the gain of this filter is always less than or equal to one, the filter outputs will fit into 16-bit variables. However the intermediate term  $113*(x[0]+x[2])$  could be as large as  $113*(1023+1023) = 231,198$ , so 32-bit calculations are performed. The gain of this filter is shown in Figure 6.12.

The “ $Q$ ” of a digital notch filter is defined to be

$$Q \equiv f_c/\Delta f$$

where  $f_c$  is the center or notch frequency, and  $\Delta f$  frequency range where is gain is below 0.707 of the DC gain. For the filter in Figure 6.12 the gains at 55 and 65 Hz



are about 0.707, so its Q is 6.

**Checkpoint 6.3:** Use Figure 6.12 to compare the filter Q of Program 6.4 with the filter Q of Program 6.6. Next, compare the execution speed of the two implementations. If you wished to remove 60 Hz and pass all other frequencies, which filter would you choose?

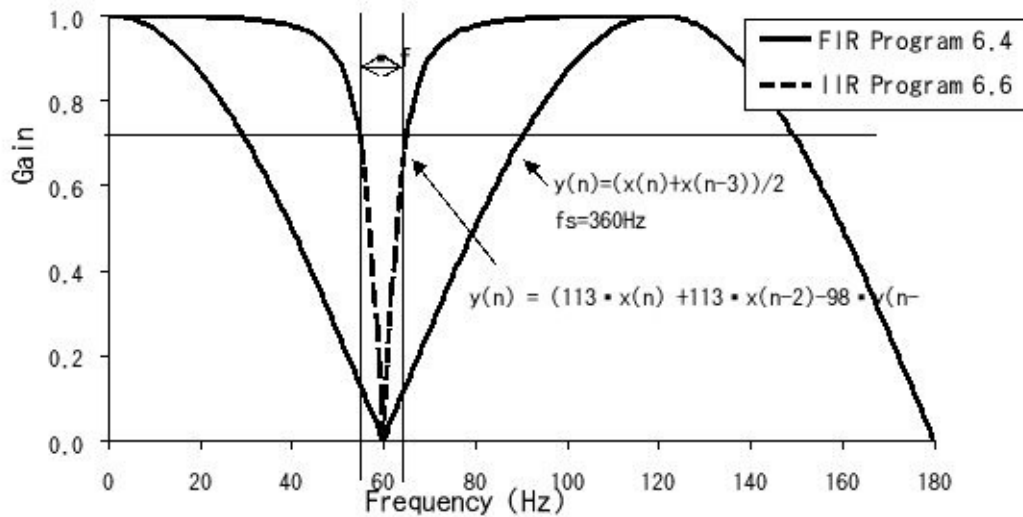


Figure 6.12. Gain versus frequency response of two 60 Hz digital notch filters.

In this second example, we will design a band-pass filter that passes 50 to 100 Hz. In this example, signals exist from 0 to 240 Hz, so the sampling rate will be 480 Hz. Figure 6.13 shows one possible pole-zero plot. First, we place the zeros so 50 to 100 Hz is passed and other frequencies are rejected. As we increase the number of zeros, we can reduce the gain in places we wish to make the gain low, but the complexity of the filter increases. This filter with 8 zeros will have 8  $x(n-k)$  terms in the equation. The idea is not to place any zeros in 50 to 100 Hz range, but place them around in the 0 to 50, and 100 to 240 regions. On the web site, there is a spreadsheet (**DigitalFilterDesign.xls**) that you can manipulate to see how the filter shape responds to the placement of poles and zeros.

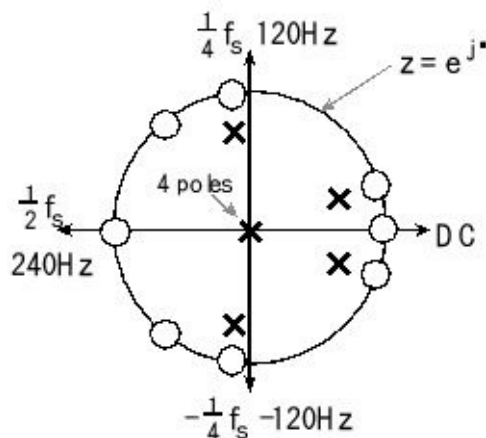


Figure 6.13. Pole-zero plot of a 50 to 100 Hz digital band-pass filter.

Next, we place the poles. In this example, there will also be 8 poles. Placing the pole near a zero causes the gain to rise and fall quickly. Placing the pole away from a zero flattens the response. In this example, the zeros near 50 and 100 Hz have poles near them, and the others are away. The farthest away will be to place the poles at  $z=0$ . The transfer function is

$$H(z) = \frac{(z-z_0)(z-z_1)(z-z_2)(z-z_3)(z-z_4)(z-z_5)(z-z_6)(z-z_7)}{(z-p_0)(z-p_1)(z-p_2)(z-p_3)(z-p_4)(z-p_5)(z-p_6)(z-p_7)}$$

The steps to derive the filter are the same as the last example. First, we multiply out the top and bottom expressions. Because the zeros are either at  $z=1$ ,  $z=-1$ , or occur in complex conjugate pairs, the numerator will have real coefficients. Similarly, because the poles are either at  $z=0$  or occur in complex conjugate pairs, the denominator will also have real coefficients. Next, we multiply the top and bottom by  $z^8$ , placing the transfer function in standard form. Next, we take the inverse transform to get the digital filter:

$$y(n) = a_0 \cdot x(n) + a_1 \cdot x(n-1) + a_2 \cdot x(n-2) + a_3 \cdot x(n-3) + a_4 \cdot x(n-4) + a_5 \cdot x(n-5) \\ + a_6 \cdot x(n-6) + a_7 \cdot x(n-7) + a_8 \cdot x(n-8) + b_0 \cdot y(n-1) + b_1 \cdot y(n-2) + b_2 \cdot y(n-3) + b_3 \cdot y(n-4)$$

Figure 6.14 plots the gain of this filter. The details of these calculations can be found in the spreadsheet **DigitalFilterDesign.xls**. The coefficients are converted to binary fixed-point and implemented in Program 6.7.

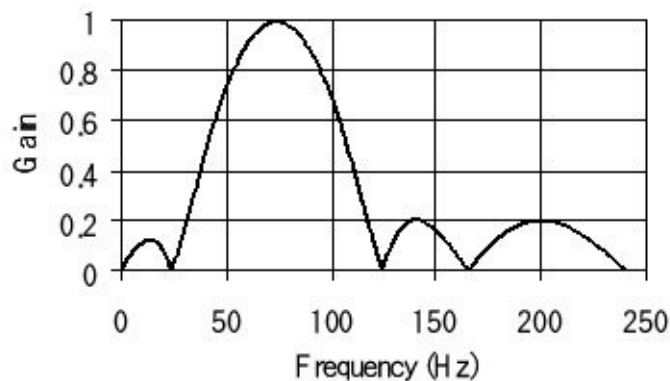


Figure 6.14. Gain versus frequency of a 50 to 100 Hz digital band-pass filter.

Typically, we design an IIR filter with an equal number of poles and zeros. If there are more zeros than poles, then filter is noncausal. For example,  $H(z)=z$  has one zero and no poles. The filter will be  $y(n) = x(n+1)$ , which is noncausal. If there are more poles than zero, then filter will have a time delay or a very large gain. For example,  $H(z)=z^{-1}$  has one pole and no zeros. The filter will be  $y(n) = x(n-1)$ .

```
const int32_t a[9]={2521,-1589,-617,-2296,0,2296,617,1589,-2521};
```

```

const int32_t b[4]={20220,-14068,9908,-3934};
int32_t x[9]; // MACQ for the ADC input data
int32_t y[5]; // MACQ for the digital filter output

void ADC3_Handler(void){
    ADC_ISC_R = 0x08; // acknowledge ADC sequence 3 completion
    x[8] = x[7]; x[7] = x[6]; x[6] = x[5]; x[5] = x[4];
    x[4] = x[3]; x[3] = x[2]; x[2] = x[1]; x[1] = x[0]; // shift data
    y[4] = y[3]; y[3] = y[2]; y[2] = y[1]; y[1] = y[0];
    x[0] = ADC_SSFIFO3_R; // 0 to 4095
    y[0] = (a[0]*x[0]+ a[1]*x[1]+ a[2]*x[2]+ a[3]*x[3]+ /* a[4]*x[4]+ */
            a[5]*x[5]+ a[6]*x[6]+ a[7]*x[7]+ a[8]*x[8]+
            b[0]*y[1]+ b[1]*y[2]+ b[2]*y[3]+ b[3]*y[4])/16384;
    Fifo_Put((int16_t)y[0]); // pass to foreground
}

```

*Program 6.7. If the sampling rate is 480 Hz, this bandpass filter passes 50 to 100 Hz.*

## 6.5. Discrete Fourier Transform

The **Discrete Fourier Transform** (DFT) converts data in the time domain to data in the frequency domain. We can use the DFT to measure SNR, to identify noise type, and to design FIR digital filters. In fact, the spectrum analyzer is simply a high-speed data acquisition system followed by a DFT. The Fast Fourier Transform (FFT) is a technique to calculate the DFT with fewer additions and multiplications. There are four important parameters when employing the DFT. The first parameter is sampling rate,  $f_s$ . While the DFT deals only with samples and bins with no concept of volts, seconds, and Hz, when applying it to real data, we assume the samples have units, are bound by physical limits, and are evenly spaced at time intervals  $T=1/f_s$ . The second parameter is sequence length,  $N$ . The other two parameters are input resolution and range. In real systems, input data come from the ADC or input capture, and the output data go to the DAC or PWM. Therefore, the performance of the DFT will be affected by the range and resolution of the input. The input to the DFT will be  $N$  samples versus time, and the output will be  $N$  points in the frequency domain.

Input:  $\{a_n\} = \{a_0, a_1, a_2, \dots, a_{N-1}\}$

Output:  $\{A_k\} = \{A_0, A_1, A_2, \dots, A_{N-1}\}$

The definition of the DFT is

$$A_k = \sum_{n=0}^{N-1} a_n W_N^{kn}$$

where

$$W_N = e^{-j2\pi/N}$$

and  $k=0,1,2,\dots,N-1$

The DFT output  $A_k$  at index  $k$  represents the amplitude and phase of the input at frequency  $k*f_s/N$  (in Hz). The DFT resolution in Hz/bin is the reciprocal of the total time spent gathering time samples; i.e.,  $1/(N*T)$ . The **Inverse Discrete Fourier Transform** (IDFT) converts data in the frequency domain to data in the time domain. The input to the IDFT will be  $N$  points in the frequency domain, and the output will be  $N$  samples in the time domain.

Input:  $\{A_k\} = \{A_0, A_1, A_2, \dots, A_{N-1}\}$

Output:  $\{a_n\} = \{a_0, a_1, a_2, \dots, a_{N-1}\}$

The definition of the IDFT is

$$a_n = \frac{1}{N} \sum_{k=0}^{N-1} A_k W_N^{-kn}$$

where

$$W_N = e^{-j2\pi n}$$

and  $n=0,1,2,\dots,N-1$

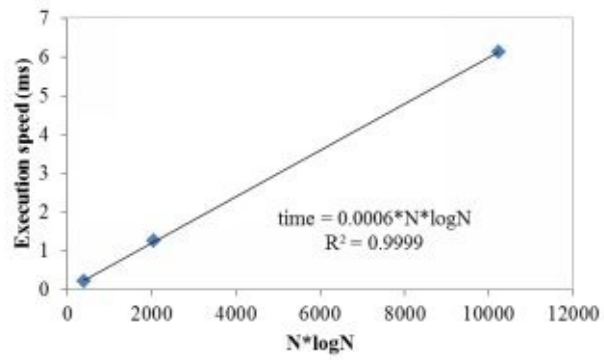
When presenting frequency data, we can use a log scale, making it easier to visualize frequency components with widely varying amplitudes. Because the system has physical limits, we use those limits to define full scale. Assume the audio system in Section 5.1.3 samples sound as a voltage from 0 to 3 V. For this system, we would define full scale  $V_{FS}$  as 3 V. In particular, if  $V$  is a DFT output in volts, we can convert it to **dB full scale** using

$$dB_{FS} = 20 * \log_{10}(V/V_{FS})$$

STMicroelectronics published integer FFT code has part of their STM32F10x\_DSP\_Lib library. There are three separate FFT implementations for sizes 64, 256 or 1024 optimized for the Cortex M. The input to the FFT is 64, 256 or 1024 complex samples. Each input is 16-bit signed integer containing the real and imaginary parts. For most applications we will set the ADC data into the real part and we will write zeros into the imaginary part. In Program 6.8 and Table 6.2 assume we will fill the input array with constant data from an array. After calculating the DFT, the program will calculate the magnitude at each frequency. Let  $N$  be the size of the array, and assume the sampling rate is  $f_s$ , then the meaning of index  $k$  is  $f_s/N$ .

```
typedef struct{
    int16_t real,imag;
}Complex_t;
// data for FFT
Complex_t x[1024],y[1024]; // input and output arrays for FFT
int32_t mag[512]; // magnitude versus frequency of the output
void cr4_fft_1024_stm32(Complex_t *, Complex_t *, unsigned short);
int main(void){ int32_t t,k, real, imag;
    for(t=0; t<1024; t=t+1){ // t means 1/fs
        x[t].imag = 0; // imaginary part is zero
        x[t].real = sinewave[t]; // fill real part with data
    }
    cr4_fft_1024_stm32(y, x, 1024); // complex FFT of 1024 values
    for(k=0; k<512; k=k+1){ // k means fs/1024
        real = y[k].real; // real is bottom 16 bits
        imag = y[k].imag; // imag is top 16 bits
        mag[k] = Sqrt(real*real+imag*imag);
    }
    while(1){};
}
```

*Program 6.8. Calculation of the FFT (ProfileFFTxxx).*



N	Cycles	Time(ms)
64	3535	0.22
256	20072	1.25
1024	97870	6.12

**Table 6.2. Execution time of the FFT varies with  $N * \log_2(N)$**

---

## 6.6. FIR Filter Design

In this section we will use the DFT as a general tool to design FIR filters. We begin by choosing the sampling rate, which must be larger than two times the largest signal frequency we wish to process. After we have chosen the sampling rate (e.g., 10 kHz), we will choose a FIR filter length (e.g.,  $N=51$ ). The ratio  $f_s/N$  (e.g.,  $10 \text{ kHz}/51 = 196 \text{ Hz}$ ) will determine the frequency resolution of the FIR filter design. The larger the  $N$ , the more gain points we can specify in the filter response, but the slower the filter will execute. Next, we plot or print the desired gain/phase versus frequency response. The magnitude of  $H(k)$  is selected to implement the desired gain versus frequency response. I.e.,  $|H(k)|$  will be the filter gain at  $k \cdot f_s/N$ . The angle of  $H(k)$  is selected to implement the desired phase versus frequency response. I.e.,  $\text{angle}[H(k)]$  will be the filter phase at  $k \cdot f_s/N$ . For frequencies above  $1/2 f_s$ , we must make  $H(k)$  be the complex conjugate of the  $N-k$  term. This will guarantee that the inverse DFT of  $H(k)$  will yield real results. Let  $x(n)$  be the input (read from the ADC) and  $X(k)$  be the input in the frequency domain. Let  $y(n)$  be the FIR filter output, and let  $Y(k)$  be the FIR filter output in the frequency domain.

$$Y(k) = H(k) X(k)$$
$$y(n) = \text{IDFT} \{ H(k) \text{DFT}\{x(t)\} \}$$

We take IDFT of the  $H(k)$  to get  $N$  FIR filter coefficients. Multiplication in the frequency domain is equivalent to **convolution** in the time domain. The FIR filter is the convolution of the data with the inverse transform of the desired filter.

$$y(n) = h(n) * x(n) = x(n) * h(n) \quad ; \quad * \text{ means convolution here}$$
$$y(n) = \text{sum}(h(i) \cdot x(n-i)) \text{ for } i=-\infty \text{ to } +\infty; \quad \cdot \text{ means multiplication here}$$

Because there are a finite number of  $h(n)$  terms, the convolution is a finite sum

$$y(n) = \text{sum}(h(i) \cdot x(n-i)) \text{ for } i=0 \text{ to } N-1; \quad \cdot \text{ means multiplication here}$$

---

**Example 6.1.** Design a digital filter for a hearing aid that accentuates high frequencies. The input is audio with frequency components from 100 Hz to 5 kHz. In particular, make the gain equal to 5 for frequencies 2 to 5 kHz. For the lower audio frequencies make the gain equal to 1.

**Solution:** We choose the sampling rate at twice the maximum frequency of the input or  $f_s = 10 \text{ kHz}$ . Next we choose a filter size. The larger  $N$ , the better the actual filter will match our desired response, but the slower it will execute. For this solution, we could have chosen any size from 32 to 64 and obtained similar results. In order to preserve the shape of the audio signals, we will implement linear phase. The desired

filter gain is shown as Figure 6.15 and Table 6.3. The lines in the figure are the desired filter gain, and the dots will be the actual gain as implemented by the fixed-point math in Program 6.9.

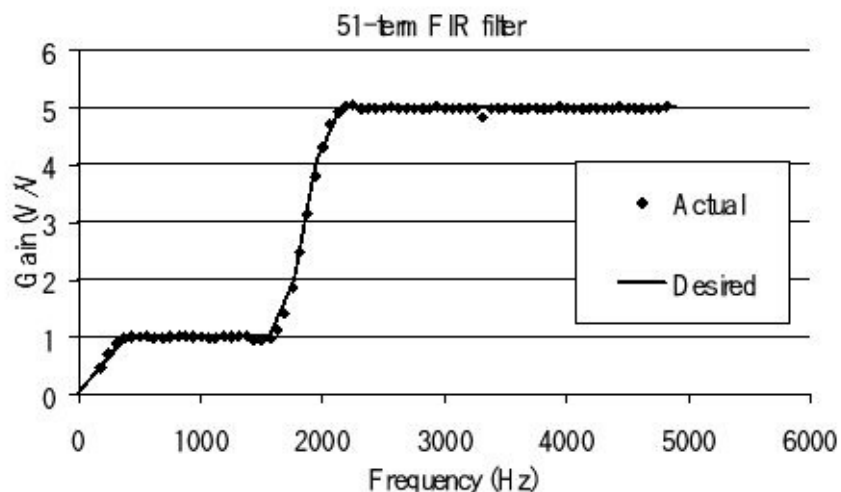


Figure 6.15. Desired and actual filter responses. This is  $H$ .

The  $H(N-k)$  values must be the complex conjugates of  $H(k)$ . Because the negative frequencies in Table 6.3 are complex conjugates of the positive frequencies,  $h(n)$  will be real. Next, we scale the  $h(n)$  values to make 51 fixed-point coefficients  $\mathbf{h}[51]$ . For example, the first term  $h(1)$  is  $-0.000457$ , which will be approximated in fixed-point as  $-7/16384$ . In summary, the  $\mathbf{h}[51]$  coefficients are the IDFT of the values in Table 6.2 multiplied by 16384 and rounded to an integer.

```
const int32_t h[51]={0,-7,-45,-64,5,78,-46,-355,-482,-138,329,
177,-722,-1388,-767,697,1115,-628,-2923,-2642,1025,4348,1820,-8027,
-19790,56862,-19790,-8027,1820,4348,1025,-2642,-2923,-628,1115,697,
-767,-1388,-722,177,329,-138,-482,-355,-46,78,5,-64,-45,-7,0};
```

k	f (Hz)	Mag(H(k))	Angle(H(k))	k	f (Hz)	Mag(H(k))	Angle(H(k))
0	0.00	0.00	0.00	13	2549.02	5.00	-40.04
1	196.08	0.50	-3.08	14	2745.10	5.00	-43.12
2	392.16	1.00	-6.16	15	2941.18	5.00	-46.20
3	588.24	1.00	-9.24	16	3137.25	5.00	-49.28
4	784.31	1.00	-12.32	17	3333.33	5.00	-52.36
5	980.39	1.00	-15.40	18	3529.41	5.00	-55.44
6	1176.47	1.00	-18.48	19	3725.49	5.00	-58.52
7	1372.55	1.00	-21.56	20	3921.57	5.00	-61.60
8	1568.63	1.00	-24.64	21	4117.65	5.00	-64.68
9	1764.71	2.00	-27.72	22	4313.73	5.00	-67.76
10	1960.78	4.00	-30.80	23	4509.80	5.00	-70.84
11	2156.86	5.00	-33.88	24	4705.88	5.00	-73.92
12	2352.94	5.00	-36.96	25	4901.96	5.00	-77.00



---

**Table 6.3. Desired filter response. This is H.**

Program 6.9 shows an implementation of this FIR filter. There are 100  $\mu$ s for each sample (ADC, filter, and DAC). We will implement the MACQ using two copies of the data, similar to Program 6.3. We could add this filter to the audio system developed in Program 5.1.

```
int16_t Data[102]; // two copies
int16_t *Pt; // pointer to current
void Filter_Init(void){
    Pt = &Data[0];
}
// calculate one filter output
// called at sampling rate
// Input: new ADC data
// Output: filter output, DAC data
int16_t Filter_Calc(int16_t newdata){
    int i; int32_t sum; int16_t *pt,*apt;
    if(Pt == &Data[0]){
        Pt = &Data[50]; // wrap
    } else{
        Pt--; // make room for data
    }
    *Pt = *(Pt+51) = newdata; // two copies
    pt = Pt; // copy of data pointer
    apt = h; // pointer to coefficients
    sum = 0;
    for(i=51; i; i--){
        sum += (*pt)*(*apt);
        apt++;
        pt++;
    }
    return sum/16384;
}
```

*Program 6.9. 51-term FIR filter*

---

**Checkpoint 6.4:** How can we prove the software in Program 6.9 cannot overflow?

**Checkpoint 6.5:** Can you think of a way to reduce the number of multiplies in Program 6.9 while still performing the exact same filter?

## 6.7. Direct-Form Implementations.

The general form for the transfer function for an IIR filter is

$$H(z) = \frac{Y(z)}{X(z)} = \frac{a_0 + a_1z^{-1} + a_2z^{-2} + \dots + a_Mz^{-M}}{b_0 + b_1z^{-1} + b_2z^{-2} + \dots + b_Nz^{-N}}$$

This converts to the standard difference equation

$$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) + \dots + a_Mx(n-M) - b_1y(n-1) - b_2y(n-2) \dots -b_Ny(n-N)$$

The direct-form calculation of this filter requires with two multiple access circular queues with lengths M and N. There are (M+N-1) multiplies and (M+N-2) additions. Figure 6.16 flow picture illustrates the standard implementation.

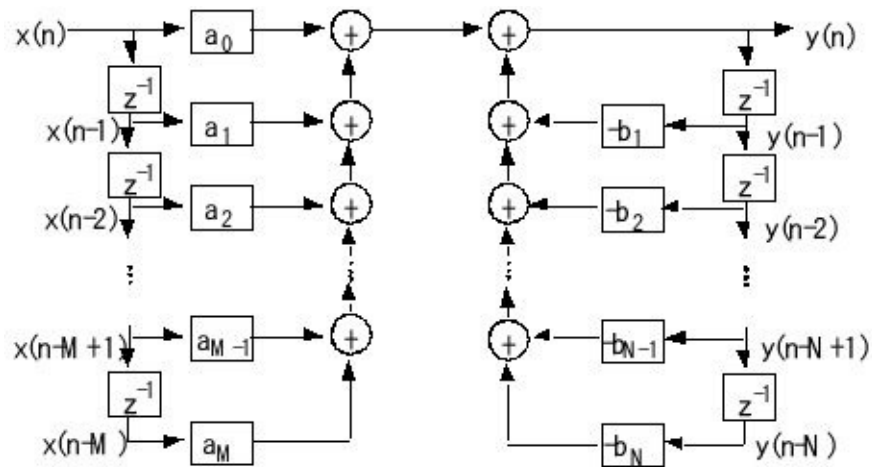


Figure 6.16. General filter design using a direct-form calculation.

For the next implementation we specify the filter with  $N=M$ . We can do this without loss of generality by letting some of the coefficients be zero. An alternative implementation, called the *direct-form II realization*, requires only one multiple access circular queue of length N. There are still (2N-1) multiplies and (2N-2) additions. Figure 6.17 illustrates the implementation.

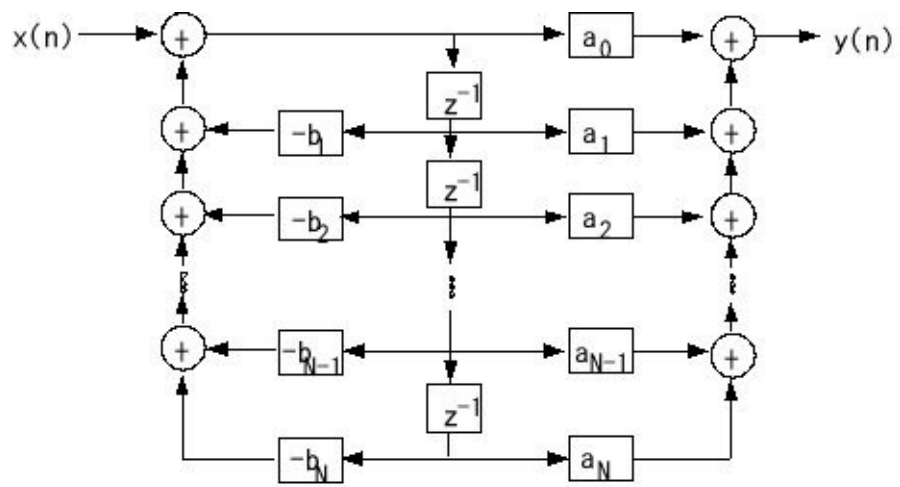


Figure 6.17. General filter design using a direct-form II calculation.

---

## 6.8. Exercises

**6.1** For each term give a definition in 32 words or less.

- a) Aliasing
- b) Filter Q
- c) Impulse response of a digital filter
- d) Complex conjugate
- e) MACQ
- f) Overflow

**6.2** For each term give the equation definition

- a) Z transform
- b) DFT
- c) IDFT
- d) Relationship between time jitter and voltage error
- e) Filter gain given input frequency  $f$
- f) Convolution between  $x$  and  $h$

**6.3.** Consider the use of the Z-transform in the design and analysis of digital filters.

- a) State the definition of the Z-transform.
- b) Why can't we use the Z-transform on a median filter?
- c) Use the Z-transform to determine the DC gain and phase of the following digital filter:

$$y(n) = x(n) - x(n-2) + y(n-1)$$

**6.4** List the four parameters we need to decide when implementing a DFT.

**6.5** For each pair of terms compare and contrast in 32 words or less.

- a) Causal versus noncausal filter
- b) Linear versus nonlinear filter
- c) FIR versus IIR filter
- d) Laplace transform versus the Z transform
- e) A pole versus a zero
- f) A complex versus an imaginary number

**6.6** 256 data points are sampled at 10 kHz with a 12-bit ADC. The ADC range is 0 to 3.0 V. A DFT is performed on the data. What is the frequency resolution? What range of frequencies is represented in the DFT output?

**6.7** For each situation, specify whether you expect the gain at frequency  $f$  to increase, decrease or not change much at all.

- a) A zero is moved closer to frequency  $f$  on the z-plane.

- b) A pole is moved closer to frequency  $f$  on the  $z$ -plane.
- c) A zero already near frequency  $f$  on the  $z$ -plane is replaced with a double zero.
- d) A pole already near frequency  $f$  on the  $z$ -plane is replaced with a double pole.
- e) A pole currently near frequency  $f$  on the  $z$ -plane is moved to the origin.
- f) A pole currently near frequency  $f$  on the  $z$ -plane is outside the unit circle.

**6.8** For each filter specify whether it is linear or nonlinear. If it is linear specify whether it is FIR or IIR.

- a)  $y(n) = x(n)^2 + 2x(n) + 1$
- b)  $y(n) = x(n)/4 + y(n-1) - x(n-4)/4$
- c)  $y(n) = \min\{x(n), x(n-1)\}$
- d)  $y(n) = (x(n+1) + x(n-1))/2$

**6.9** Let the input be the sum of two sine waves:  $x(t) = A_1 \sin(2\pi f_1 t) + A_2 \sin(2\pi f_2 t)$ . Assume the digital filter will pass both these frequencies with a gain of 1. This filter implements a linear phase response. What can you say about the output of the filter? I.e., derive an equation describing the output as a function of time.

**6.10** Consider the following digital filter:  $y(n) = (x(n) - x(n-2))/2$

- a) Using the Z-transform derive general expressions for the gain and phase of the filter.
- b) Using the general expressions from part a), calculate the gain and phase of the filter at DC and 60 Hz if the sampling rate is 240 Hz.

**6.11** Design a 10 Hz digital low pass filter with a sampling rate of 1000Hz. Make the gain at DC equal to one, and the gain at 10Hz 0.707.

- a) Show the pole/zero plot of your filter.
- b) Show the  $H(z)$  transform.
- c) Show the floating-point version of the digital filter.
- d) Show the fixed-point version of the digital filter.

**6.12** Design a digital filter that rejects both 60 Hz and 120Hz assuming the sampling rate is 480 Hz. Apply gain scaling so the DC gain is 1. Give the filter in a form that can be implemented with fixed-point math.

**6.13** Consider the simple sliding average filter for a general sampling rate of 1000 Hz. This filter is a low-pass filter, as shown in Figure 6.10

$$y(n) = \frac{1}{k} \sum_{i=0}^{i=k-1} x(n-i)$$

What value of  $k$  should we use to make a gain of about 0.7 at 10 Hz?

**6.14** We defined time-jitter,  $\delta t$ , as the difference between when a periodic task is supposed to be run, and when it is actually run. The goal of a real-time DAS is to

start the ADC at a periodic rate,  $\Delta t$ . Let  $t_n$  be the  $n$ th time the ADC is started. In particular, the goal to make  $t_n - t_{n-1} = \Delta t$ . The jitter is defined as the constant,  $\delta t$ , such that

$$\Delta t - \delta t < t_i - t_{i-1} < \Delta t + \delta t \quad \text{for all } i.$$

Assume the ADC input can be described as  $V(t) = A + B\sin(2\pi ft)$ , where  $A$ ,  $B$ ,  $f$  are constants.

a) Derive an estimate of the maximum voltage error,  $\delta V$ , caused by time-jitter. Basically, solve for the largest possible value of  $\delta V$  as a function of  $\delta t$ ,  $A$ ,  $B$ , and  $f$

b) Consider the situation where this time jitter is unacceptably large. Which modification to the system will reduce the error the most? Justify your selection.

- A) Run the ADC in continuous mode
- B) Convert from spinlock semaphores to blocking semaphores
- C) Change from round robin to priority thread scheduling
- D) Reduce the amount of time the system runs with interrupts disabled.
- E) Increase the size of the DataFifo

# 7. High-Speed Interfacing

**Chapter 7 objectives are to:**

- Discuss applications requiring high bandwidth
- Present concepts related to high-speed interfacing
- List fundamental approaches to high-speed interfacing
- Introduce and describe direct memory addressing (DMA) on the TM4C123

Embedded system designers will not need direct memory accessing to solve most of their problems. However, future trends point to systems with increased memory, multiple processors and higher bandwidth. Therefore, it is appropriate to learn these advanced topics. Latency, bandwidth, synchronization, and reliability are important factors for all types of interfacing. In this chapter we will discuss shared memory, hardware FIFOs, and direct memory addressing (DMA). DMA is an important yet complicated interfacing process. As the performance requirements of our embedded system grow, there comes a point when the simple methods of I/O interfacing are not adequate. This chapter introduces a number of techniques that produce high bandwidth and low latency.

## 7.1. The Need for Speed

Bandwidth, latency, and priority are quantitative parameters we use to evaluate the performance of an I/O interface. The basic function of an input interface is to transfer information about the external environment into the computer. In a similar way, the basic function of an output interface is to transfer information from the computer to the external environment. The **bandwidth** is the number of bytes transferred per second. The bandwidth can be expressed as a maximum or peak that involves short bursts of I/O communication. On the other hand, the overall performance can be represented as the average bandwidth. The latency of the hardware/software is the response time of the interface. It is measured in different ways depending on the situation. For an input device, the **interface latency** is the time between when new input is available, and the time when the data is transferred into memory. We can also define device latency as the response time of the external I/O device. For example, if we request that a certain sector be read from a disk, then the device latency is the time it takes to find the correct track and spin the disk (seek) so the proper sector is positioned under the read head. For an output device, the interface latency is the time between when the output device is idle, and the time when the interface writes new data. A **real-time** system is one that can guarantee worst case interface latency. Table 7.1 illustrates specific ways to calculate latency. In each case, however, latency is the time between when the need arises to the time the need is satisfied.

<i>The time a need arises</i>	<i>The time the need is satisfied</i>
New input is available	The input data is read
New input is available	The input data is processed
Output device is idle	New output data is written
Sample time occurs	ADC is triggered, input data
Periodic time occurs	Output data, DAC is triggered
Control point occurs	Control system executed

**Table 7.1. Interface latency is a measure of the response time of the computer to a hardware event.**

If we consider the busy/done I/O states, the latency is the time from busy to done state transition to the time of the done to busy state transition. Sometimes we are



interested in the worst case (maximum) latency and sometimes in the average. If we can put an upper bound on the latency, then we define the system as real-time. A number of applications involve performing I/O functions on a fixed interval basis. In a data acquisition system, the ADC is triggered (a new sample is requested) at the desired sampling rate.

**Checkpoint 7.1:** What is the difference between bandwidth and latency?

---

## 7.2. High-Speed I/O Applications

Before introducing the various solutions to a high-speed I/O interface, we will begin by presenting some typical applications.

*Mass Storage.* The first application is mass storage including flash disk, hard disk, CD, and DVD. Writing data to disk with these systems involves

1. Establishing the physical location to write, record head at the proper block, sector, track etc.
2. Specifying the block size
3. Waiting for the physical location to arrive under the record head
4. Transmitting the data

Reading data from disk with these systems is similar and involves

1. Establishing the physical location to read, read head at the proper block, sector, track etc.
2. Specifying the block size
3. Waiting for the physical location to arrive under the read head
4. Receiving the data

Under most situations the size of the data block transferred is fixed. The bandwidth depends on the rotation speed of the disk and the information density on the medium. A 10,000 RPM SATA hard drive can sustain about 157 Mebibyte/sec. However, drives costing less than \$100 typically generate 100 Mebibytes/sec. The time to locate the physical location is called the seek time. Although seek time has a significant impact on the disk performance, it does not affect the latency or bandwidth parameters. An  $nX$  CD-ROM has a peak bandwidth of  $n*150$  kibibytes/sec. There is a wide range of disk speeds, but it is important to note that for most situations, the disk bandwidth will be less than the computer bus bandwidth, but greater than the maximum bandwidth that a software-controlled interface can achieve. If the disk interface is not buffered, then the interface must respond to each data byte at a rate faster than the peak disk bandwidth. For example, in a disk read, once the data becomes available, the interface must capture it and store it in memory before the next data becomes available. If we do not meet the response time requirement in the disk interface, the rotation speed will have to be reduced. Notice because of the seek time (time for the physical location to arrive under the head), the average and peak bandwidth will be quite different. Also notice that without buffering, the maximum interface latency will be inversely related to the peak bandwidth.

**Checkpoint 7.2:** What happens if we are reading data off a hard drive but do not satisfy the latency requirement? In other words, the read data is ready, but we do not capture it in time.

*High-Speed Data Acquisition.* Examples of high-speed data acquisition are CD-

quality sound recording (16-bit, 2 channel, 44 kHz), real-time digital image recording and digital scopes (8-bit 1 GHz). Sound recording actually has two high-speed data channels: one for recording into memory, and a second for storing the memory data on hard disk or CD. Similarly, a digital scope has two high-speed channels: one for the recording of voltage versus time input, and a second for displaying graphical results. A spectrum analyzer combines the high-speed data acquisition of a digital scope with the discrete Fourier Transform to visualize the collected data in the frequency domain. In the context of this chapter, we will define a high-speed data acquisition as one that samples faster than a software-controlled interface would allow. Typically, this will mean more than 100,000 samples per second.

**Checkpoint 7.3:** What happens to the sound recording if data is missed? Is this hard, firm, or soft real time?

*Video displays.* Real-time generation of TV or video images requires an enormous data bandwidth. Consider the information bandwidth required to maintain an image on a graphics display. A VGA image is 256 colors (8-bit), 480 rows, 640 columns and is refreshed at about 60 Hz. Calculating the bandwidth in bytes/sec, we get  $1 \times 480 \times 640 \times 60$ , which is 18,432,000 bytes/sec. Luckily, we don't have to communicate each pixel for each image, but rather can just transmit the changes from the previous image. In order to achieve the necessary bandwidth, video interface hardware will use a combination of DMA and dual port memories. With larger displays and 3-D images the bandwidth requirements are even higher.

*High Speed Signal Generation.* Examples of high speed signal generation are CD-quality sound playback (16-bit, 2 channel, 44 kHz) and real-time waveform generation. Sound playback also has two high speed data channels: one for loading sound data into memory from CD, and a second for playing the memory data out to the speakers.

*Network Communications.* For many networks the communication bandwidth of the physical channel will exceed the ability of the software to accept or transmit messages. For these high speed applications, we will look for ways to decouple the software that creates outgoing messages and processes incoming messages from the hardware that is involved in the transmission and reception of individual bits. Because the network load will vary, the average bandwidth (determined by how fast the transmission software can create outgoing messages and the reception software can process incoming messages) will be slower than the peak/maximum bandwidth that is achieved by the network hardware during transmission. This mismatch allows one network to be shared among multiple potential nodes.

**Checkpoint 7.4:** What happens in a communication system when packets are lost?

## 7.3. General Approaches to High-Speed Interfaces

### 7.3.1. Hardware FIFO

If the software-controlled interface can handle the average bandwidth but fails to satisfy the latency requirements, then a **hardware FIFO** can be placed between the I/O device and the computer. Assume in this situation, the average serial bandwidth is low enough for the software to read the data from the serial port and write it to memory. Without the hardware FIFO, the latency requirement of a serial input port is the time it takes to transmit one data frame. To reduce this latency requirement (without changing the average bandwidth requirement) we can add a hardware FIFO between the receive shift register and the receive data register, as illustrated in Figure 7.1. Many of the I/O devices on the Texas Instruments microcontrollers employ hardware FIFOs.

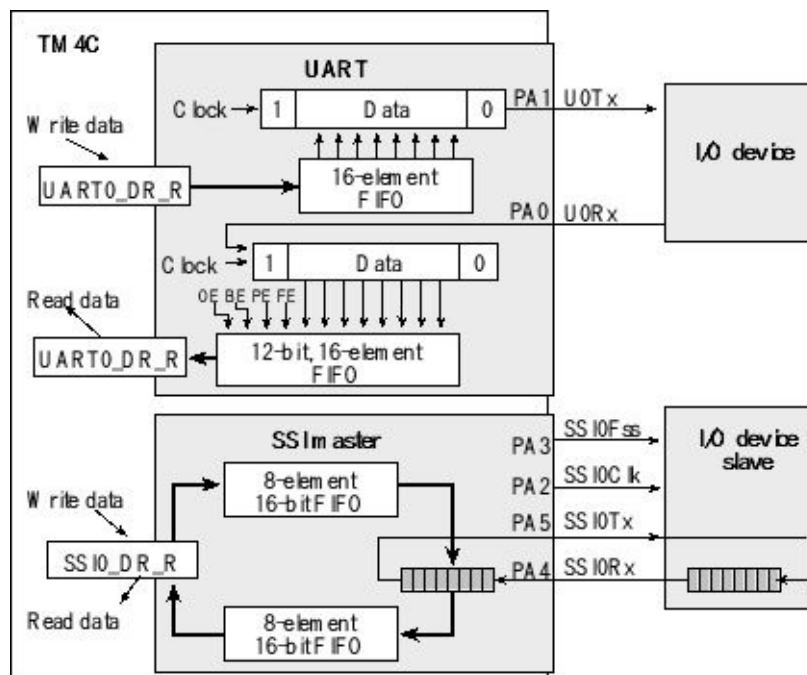


Figure 7.1. High-speed I/O devices employ hardware FIFOs to reduce the latency requirement of the interface.

**Observation:** With a serial port that has a shift register, a FIFO of size  $n$ , and one data register, the latency requirement of the input interface is the time it takes to transmit  $n+1$  data frames.

A hardware FIFO, placed between the output data register and the transmit shift register, allows the software to write multiple bytes of data to the interface and then perform other tasks while the frames are being sent.

## 7.3.2. Dual Port Memory

One approach that allows a large amount of data to be transmitted from the software to the hardware is the dual port memory, Figure 7.2. A dual port memory allows shared access to the same memory between the software and hardware. For example, the software can create a graphics image in the dual port memory using standard memory write operations. At the same time the video graphics hardware can fetch information out of the same memory and display it on the computer monitor. In this way, the data need not be explicitly transmitted from the computer to the graphics display hardware. To implement a dual port memory, there must be a way to arbitrate the condition when both the software and hardware wish to access the device simultaneously. One mechanism to arbitrate simultaneous requests is to halt the processor using a MRDY signal so that the software temporarily waits while the video hardware fetches what it needs. Once the video hardware is done, the MRDY signal is released and the software resumes. Most microcontroller memory interfaces do not support this sort of hardware initiated cycle stretching. If both processors wish to access the memory at the same time, one of the processors is delayed. Notice that except for the access conflict, both the software and graphics hardware can operate simultaneously at full speed.

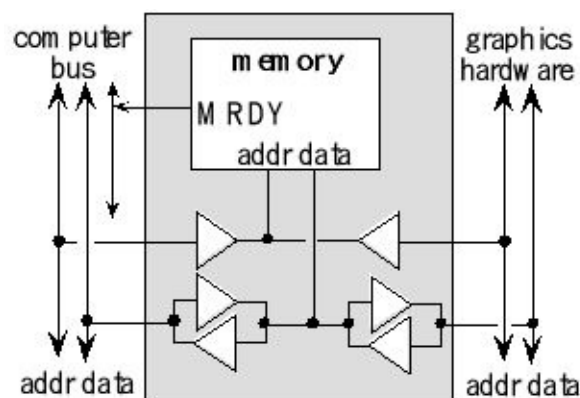


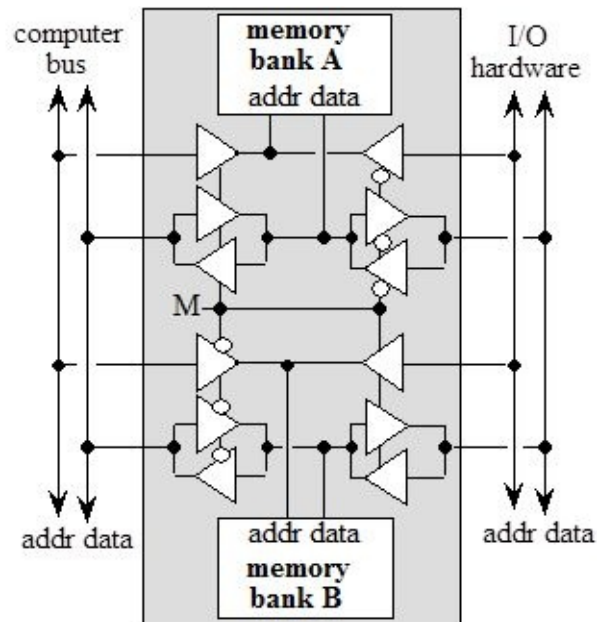
Figure 7.2. A dual port memory can be accessed by two different modules.

**Checkpoint 7.5:** Explain how the bidirectional tristate buffers connected to the memory data lines in Figure 7.2 work.

## 7.3.3. Bank-Switched Memory

Another approach similar to the dual port memory is the bank-switched memory, see Figure 7.3. A bank-switched memory also allows shared access to the same memory between the software and hardware. The difference between bank-switched and dual port is the bank-switched memory has two modes. In one mode ( $M=1$ ), the computer has access to memory bank A, and the I/O hardware has access to memory bank B. In the other mode ( $M=0$ ), the computer has access to memory bank B, and the I/O

hardware has access to memory bank A. Because access is restricted in this way, there are no conflicts to resolve.



*Figure 7.3. A bank-switched memory can be accessed by two different modules, one at a time.*

**Observation:** With a bank-switched memory, the latency requirement of the software is the time it takes the hardware to fill (or empty) one memory bank.

Graphics controllers use bank switching. One processor transfers data from the front buffer and displays it on the screen. A second processor builds the next image in the back buffer. To create the video output, the buffers are switched at a regular rate. Many high speed data acquisition systems all employ bank switching. The ADC hardware can write into one bank while the computer software processes previously collected data in the other. When the ADC sampling hardware fills a bank, the memory mode is switched, and the software and hardware swap access rights to the memory banks. In a similar way, a real-time waveform generator or sound playback system can use the bank-switched approach. The software creates the data and stores it into one bank, while the hardware reads data from the other bank that was previously filled. Again, when the hardware is finished, then the memory bank mode is switched.

**Checkpoint 7.6:** How would you redesign the bank-switched memory in Figure 7.3 if the communication channel were simplex (data flows left to right only)?

## 7.4. Fundamental Approach to DMA

With a software-controlled interface (busy-wait or interrupts) if we wish to transfer data from an input device into RAM, you must first transfer it from input to the processor, then from the processor into RAM. In addition, this transfer is explicitly controlled by executing software. In order to improve performance, we will transfer data directly from input to RAM or RAM to output using **Direct Memory Access**, DMA. Because DMA bandwidth can be as high as the bus bandwidth, we will use this method to interface high bandwidth devices like disks, digital scopes, cameras, and networks. Similarly, because the latency of this type of interface depends only on hardware and is usually just a couple of bus cycles, we will use DMA for situations that require a very fast response. On the other hand, software-controlled interfaces have the potential to perform more complex operations than simply transferring the data to/from memory. For example, the software could perform error checking, convert from one format to another, implement compression/decompression, and detect events. These more complex I/O operations may preclude the usage of DMA.

### 7.4.1. DMA Cycles

During a DMA read cycle, the processor can still access flash memory and ROM, while hardware automatically transfers data from RAM to the output device (Figure 7.4). The address on the bus specifies the RAM location from which to read the data. The  $\mu$ DMA controller on the TM4C has many different configuration options to burst transfer data to and from arbitrary locations. For example, it may automatically increment the RAM source address to stream an array to an output device. The TM4C series does not support DMA transfers with flash memory or ROM because they are on a separate internal bus.

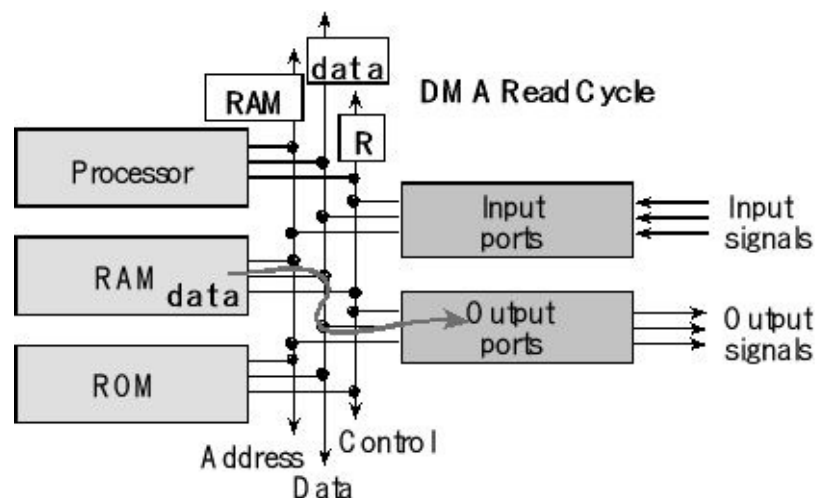


Figure 7.4. A DMA read cycle copies data from RAM to an output device.

During a DMA write cycle, the processor can still access flash memory and ROM, while hardware automatically transfers data from the input device to RAM (Figure 7.5). The address on the bus specifies the RAM location to which to write the data. A useful configuration mode could be to have the  $\mu$ DMA controller automatically increment the RAM destination address to stream data from an input device. In some DMA interfaces, two DMA cycles are required to transfer the data. The first DMA cycle brings data from the source into the DMA module, and the second DMA cycle sends the data to its destination.

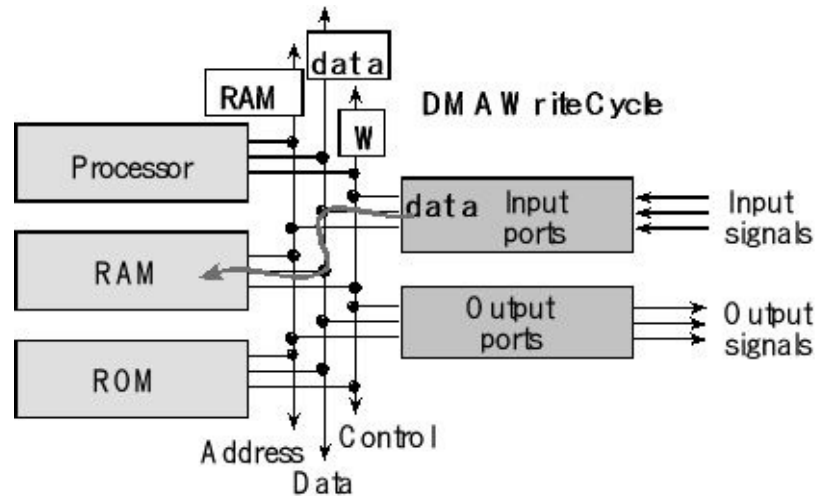


Figure 7.5. A DMA write cycle copies data from the input device into RAM.

## 7.4.2. DMA Initiation

We can classify DMA operations according to the event that initiates the transfer. A **software initiated** transfer begins with the program setting up and starting the DMA operation. Using DMA to transfer data from one memory block to another greatly speeds up the function. The efficiency of memory block transfers is very important in larger computer systems. Benchmarks on most computers show that even for small blocks, it is faster to initialize a DMA channel and perform the transfer in hardware than it is to transfer the data block using software. As the block size increases the performance advantage of DMA hardware over traditional software becomes more dramatic.

Most DMA applications involve a **hardware initiated** DMA transfer. For an input device, the DMA is triggered on new data available. For an output device, the DMA is triggered on output device idle. For periodic events, like data acquisition and signal generation, the DMA is triggered by a periodic timer. These are the exact issues involved in busy-wait loop and interrupt synchronization. The difference with DMA is that the servicing of the I/O need will be performed by the DMA controller hardware without software having to explicitly transfer each byte. An interrupt is typically triggered at the end of the block transfer.



### 7.4.3. Burst versus Single Cycle DMA

When the desired I/O bandwidth matches the computer bus bandwidth, then the computer can be completely halted, while the block of data is transferred all at once, see Figure 7.6. Once an input block is ready, a burst mode DMA is requested, the computer is halted, and the block is transferred into memory.



Figure 7.6. An input block is transferred all at once during burst mode DMA.

Figure 7.6 describes an input interface, but the same timing occurs on an output interface using burst mode DMA. For an output interface, the DMA is requested when the interface needs another block of data. During the burst mode DMA, the computer is halted, and an entire block is transferred from memory to the output device.

If the I/O bandwidth is less than the computer bus bandwidth, then the DMA hardware will steal cycles and transfer the data a single cycle at a time, see Figure 7.7. In single cycle mode, the software continues to run, although a little bit slower. In either case the processor is halted during the DMA cycles.

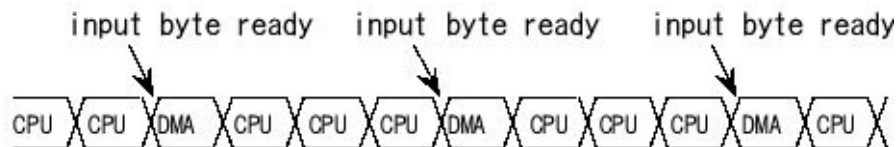


Figure 7.7. Each time an input byte is ready it is transferred to memory using single cycle DMA.

Figure 7.7 describes an input interface, but the same timing occurs on an output interface using cycle steal mode DMA. For an output interface, the DMA is requested when the interface needs another byte of data. During the single cycle DMA, one byte/halfword/word is transferred from RAM to the output device.

**Observation:** Some computers must finish the instruction before allowing a burst-DMA. In this situation, the latency will be higher than single cycle DMA, which does not need to finish the current instruction.

Since most I/O bandwidths are indeed less than the memory bandwidth, one technique to enhance speed is I/O buffering. In this approach a dedicated I/O memory buffer exists in the I/O interface hardware. This buffer is like the bank-switched memory discussed earlier. For example, on a hard disk read block operation, raw data comes off the disk and into the buffer. During this time the processor is not halted. When the buffer is full, burst DMA is used to transfer the data into the system memory. Similarly, on a hard disk write block operation, the software initiates a burst DMA to transfer data from system memory into the I/O buffer. Once full, the I/O

interface can write the data onto the disk.

**Checkpoint 7.7:** What is the maximum latency in a single cycle DMA system?

### 7.4.4. Single Address versus Dual Address DMA

Some computer systems allow the transfer of data between the memory and I/O interface to occur in one bus cycle, while others need two bus cycles to complete the transfer.

In a **single address DMA cycle**, the address and R/W line dictate the memory function to be performed and the I/O interface is sophisticated enough to know it should participate in the transfer. In this single address example, the disk interface is reading bytes from a disk, as shown in Figure 7.8. During the transfer, the bus address is the memory address, Figure 7.9.

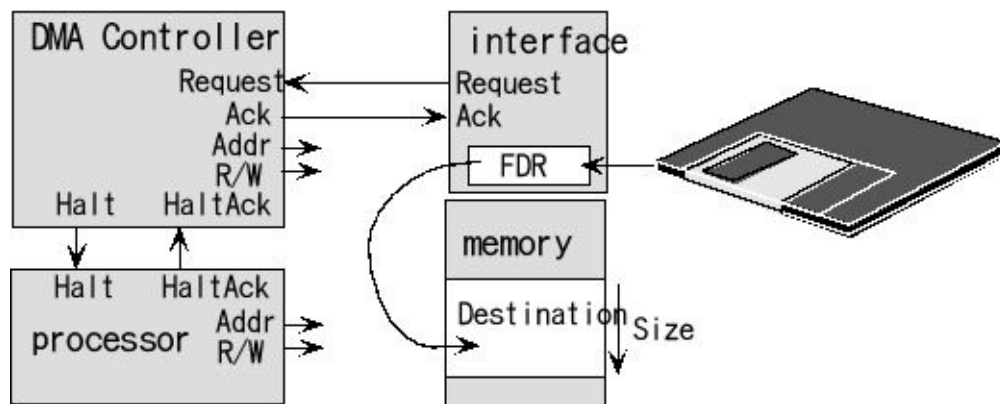


Figure 7.8. Block diagram showing the modules involved in a disk read.

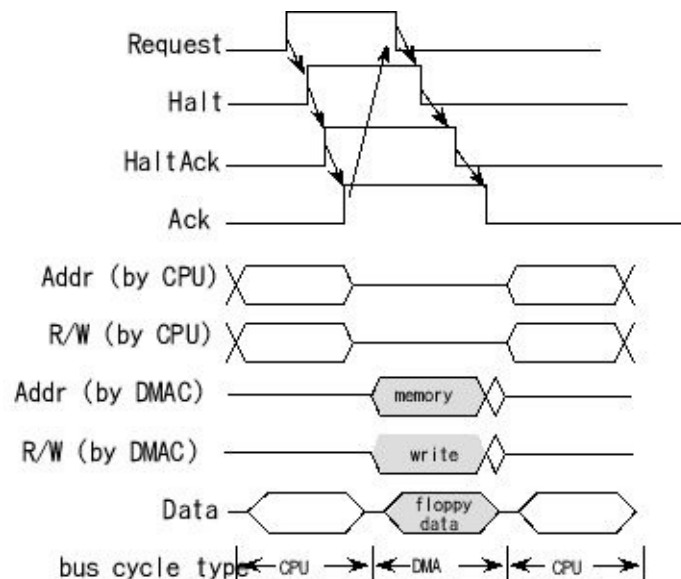


Figure 7.9. Timing diagram of a single address DMA-controlled floppy disk read.

Single cycle mode will be used because the disk bandwidth is slower than the bus. When a new byte is available, **Request** will be asserted and this will request a DMA cycle from the DMA controller (Figure 7.9). The DMA controller will temporarily suspend the processor and drive the address bus with the memory address and the R/W to write. During this cycle the DMA controller will respond to the floppy interface by asserting the **Ack**. The disk uses the **Ack** (ignoring the address bus and R/W) to know when to drive its data on the bus.

**Observation:** Most microcontrollers including the MSP432 and the TM4C do not support single address DMA.

In a **dual address DMA** cycle, two bus cycles are required to achieve the transfer. In the first cycle, the data is read from the source address and copied in the DMA controller. During the first cycle the address bus contains the source address and R/W signifies read. The information from the data bus is saved in the Temp register within the DMA controller. In the second cycle, the data is transferred to the destination address. During the second cycle, the address bus contains the destination address, the data bus has the Temp data, and R/W signifies write. In this dual address example, the SPI interface is receiving bytes from a synchronous serial network (Figure 7.10). Single cycle mode will be used because the SPI bandwidth is slower than the bus. When a new byte is available, **Request** will be asserted and this will request a DMA cycle from the DMA controller (Figure 7.11). The DMA controller will temporarily suspend the processor and first drive the address bus with the SPI data register address (R/W=read), then in the second cycle the DMA controller will drive the address bus with the memory address (R/W=write). The SPI knows it has been serviced, because its data register has been read. The single address DMA is twice as fast as dual address DMA.

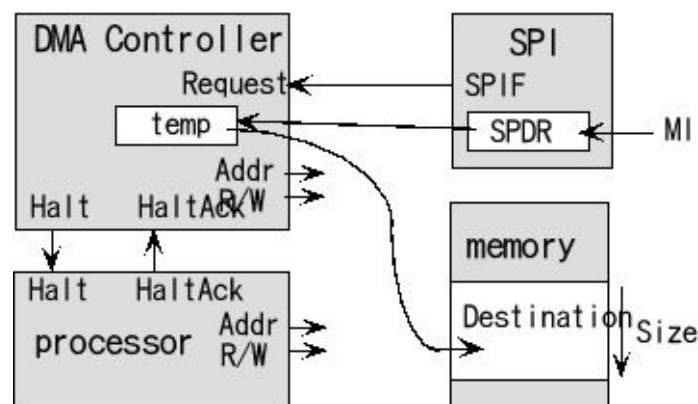


Figure 7.10. Block diagram showing the modules involved in a SPI read.

**Observation:** The dual address DMA can be used with I/O devices not configured to support DMA. Basically, we can transfer data between any I/O register and/or memory location.

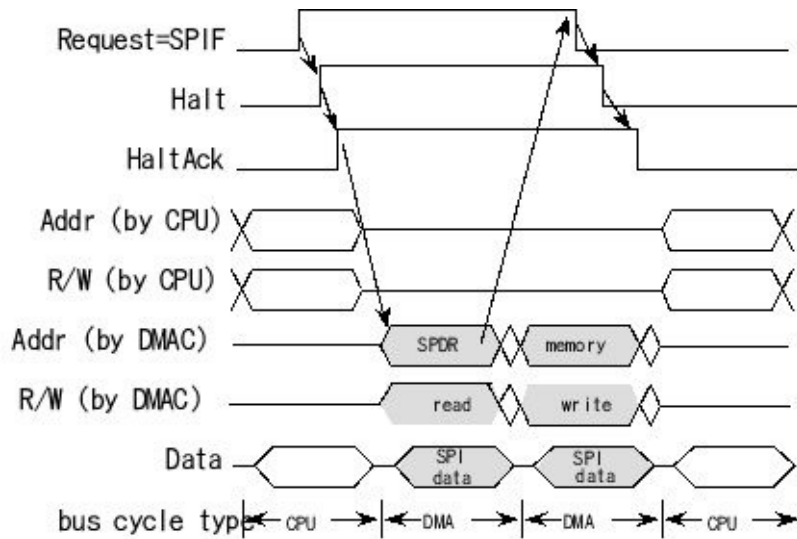


Figure 7.11. Timing diagram of a dual address DMA-controlled SPI read.

## 7.4.5. DMA programming on the TM4C123

Although DMA programming varies considerably from one system to another, there are a few initialization steps that most require. Example projects using DMA can be found in the TivaWare projects. Table 7.2 lists the mode parameters that must be set to utilize DMA. There are two categories of DMA programming: initialization and completion. During initialization, the software sets the DMA parameters, so that the DMA will begin.

Parameter	Possible choices
What initiates the DMA	Software trigger, input device, output device, periodic timer
Type	Burst versus single
Autoinitialization mode	Single event or continuous transfer
Precision	8-bit byte, 16-bit half-word or 32-bit word
Mode	Single address or dual address
Priority	Default or high priority; lower channel numbers are higher priority to break a tie
Synchronization	Set busy-wait flag, or interrupt on block transfer complete

Table 7.2. DMA initialization usually involves specifying these parameters.

At the end of a block transfer, a done flag is set and a number of additional actions may occur. If the system is armed, an interrupt can be generated. At the end of a block transfer in a continuous transfer DMA, the controller automatically switches between the primary and alternate control structures and continues transferring. At this point, a

little bit of software attention is required to allow the DMA process to continue indefinitely. An interrupt is requested, the DMA controller is finished with one control structure so that one is stopped, and it has moved on to the other control structure, which is running. In this case, software must look at the **XFERMODE** field of the DMA Channel Control Word register of both the primary and alternate control structures. If this field is zero, the corresponding control structure is stopped and must be re-initialized before the active one finishes. The TM4C calls this ping-pong mode. Table 7.3 lists additional parameters we will need to initialize.

Parameter	Definition
Source address end pointer	Last address of the module (RAM or input) that generates the data, inclusive
Destination address end ptr.	Last address of the module (RAM or output) that accepts the data, inclusive
Destination address incr.	Automatically increment the destination address by 8, 16, 32, or 0 The address increment bit field value must be $\geq$ data size bit field value
Destination data size	8-, 16-, or 32-bit data size Destination data size must be the same as source data size
Source address increment	Automatically increment the source address by 8, 16, 32, or 0 The address increment bit field value must be $\geq$ data size bit field value
Source data size	8-, 16-, or 32-bit data size Source data size must be the same as destination data size
Arbitration size	Number of DMA transfers before the controller re-arbitrates channel priority This arbitration is among DMA channels only; DMA never blocks processor Size can be thought of as the maximum burst size Should equal what peripheral can accommodate on burst request Must be a power of 2, but no arbitration occurs if $\geq 1,024$
Transfer size (minus 1)	Number of transfers to be made Maximum bit field value of 1,023 representing maximum of 1,024 transfers Updated by hardware at arbitration to contain number of transfers remaining
Next useburst	If the number of transfers remaining is less than the

	arbitration size, setting this bit uses a burst transfer to get all of them; otherwise use single transfers Used exclusively for the peripheral scatter-gather operation
Transfer mode	Configure the DMA transfer mode according to desired operation of system

**Table 7.3. DMA initialization parameters from the control structure located in RAM.**

**Checkpoint 7.8:** What is the maximum latency in a dual-address burst DMA system?

**Checkpoint 7.9:** What is the maximum bandwidth in a dual-address burst-DMA system?

The web site contains four examples of DMA transfer: RAM-RAM block transfer (**DMASoftware\_4C123**), continuous output to DAC (**DMASPI\_4C123**) effectively playing a continuous audio track, continuous input from a GPIO port (**DMATimer\_4C123**) creating a logic analyzer, and continuous output to a port (**DMATimerPortWrite\_4C123**.)

To illustrate the use of DMA a simple memory to memory block transfer will be shown. There are 32 DMA channels available on the TM4C123, and channel 30 is dedicated to software triggered memory to memory transfers. There are some configurations that occur just once, and can be placed in the initialization code. See Program 7.1. The clock is enabled in the **SYSCTL\_RCGCDMA\_R** register. The **MASTEN** bit is turned on in the **UDMA\_CFG\_R** register to activate the  $\mu$ DMA device. Configuring the DMA transfer centers around the Channel Control Structure, see Table 7.4. There is a 3-word entry in this structure for each of the 32 DMA channels. The **UDMA\_CTLBASE\_R** register is configured to point to the Channel Control Structures. The first half of the table contains 32 entries specifying the primary command for each channel and the second half is another 32 entries specifying the alternate commands. This memory to memory transfer only uses the primary command. The command entry for channel 30 exists in words 120, 121, and 122 within this table. Each entry is aligned to a 4-word boundary by skipping one word. We set bit 30 in the **UDMA\_PRIOCLR\_R** register to specify default priority. Conversely, if we were to set bit 30 in the **UDMA\_PRIOSET\_R** register, then this channel would have high priority over other DMA channels. We set bit 30 in the **UDMA\_ALTCLR\_R** register to disable the alternate control table, using just the primary entries. There are two types of DMA transfer single cycle and burst. We set bit 30 in the **UDMA\_USEBURSTCLR\_R** register to allow both single cycle and burst DMA. This example will burst 8 transfers at a time. By setting bit 30 of the **UDMA\_REQMASKCLR\_R** register we activate channel 30.

Each time a DMA transfer is started, the software must configure the three words in the  $\mu$ DMA Channel Control Structure. For each channel there are three words: source

address, destination address, and a channel control word. More specifically, we will place the addresses of the last memory locations to be transferred into the source and destination fields. There are eight fields in the control word. The **DSTINC** and **SRCINC** specify if the source and destination addresses should be incremented (0 means +1, 1 means +2, 2 means +4, and 3 means no increment). In this example we set both **DSTINC** and **SRCINC** to 2 so +4 is added to the addresses after each word is transferred. The **DSTSIZE** and **SRCSIZE** specify the data size of the source and destination (0 means byte, 1 means halfword, and 2 means word). In this example we set both **DSTSIZE** and **SRCSIZE** to 2 to specify the transfer of 32-bit data. The **ARBSIZE** field specifies the size of the bursts used during transfer. By setting this field to 3, the controller will burst 8 words then look to see if another module wishes to use the bus. The **XFERSIZE** field specifies the number of items to transfer. By setting this field to **cnt-1**, the controller will transfer **cnt** words. The **NXTUSEBURST** field is not used in memory to memory transfer. We set the **XFERMODE** bits to 2 to select auto-request mode. We set bit 30 in the **UDMA\_ENASET\_R** register to enable channel 30. By setting bit 30 of the **UDMA\_SWREQ\_R** register the transfer is initiated. There are three possible mechanisms to determine when the transfer is complete. First, when complete, bit 30 of the **UDMA\_ENASET\_R** register will become zero. Alternately, we could poll the **XFERMODE** bits in the channel control structure; these bits will also go zero when the transfer is complete. A third mechanism uses interrupts. If we arm interrupt number 46 in the NVIC, which is vector 62 at address 0x0000.00F8, then a  $\mu$ DMA Software interrupt will be generated on completion.

Address of the last byte of the source buffer								
Address of the last byte of the destination buffer								
DSTINC	DSTSIZE	SRCINC	SRCSIZE		ARBSIZE	XFERSIZE	NXTUSE	XFERMODE

**Table 7.4. Structure of an entry in the channel control structure.**

**// The ucControlTable table must be aligned to a 1024 byte boundary.**

**uint32\_t ucControlTable[256] \_\_attribute\_\_((aligned(1024)));**

**#define CH30 (30\*4)**

**#define BIT30 0x40000000**

**// \*\*\*\*\*DMA\_Init\*\*\*\*\***

**// Initialize the memory to memory transfer**

**// This needs to be called once before requesting a transfer**

**// Inputs: none**

**// Outputs: none**

**void DMA\_Init(void){**

**volatile uint32\_t delay;**

**SYCTL\_RCGCDMA\_R = 0x01; //  $\mu$ DMA Module Run Mode Clock Gating Control**

**delay = SYCTL\_RCGCDMA\_R; // allow time to finish**

**UDMA\_CFG\_R = 0x01; // MASTEN Controller Master Enable**

```

UDMA_CTLBASE_R = (uint32_t)ucControlTable;
UDMA_PRIOCR_R = BIT30; // default, not high priority
UDMA_ALTCLR_R = BIT30; // use primary control
UDMA_USEBURSTCLR_R = BIT30; // responds to both burst and single
UDMA_REQMASKCLR_R = BIT30; // allow controller to recognize requests
}
// *****DMA_Xfr *****
// Called to transfer words from source to destination
// This needs to be called once before requesting a transfer
// Inputs: src is a pointer to the first element of the original data
//         dest is a pointer to a place to put the copy
//         cnt is the number of words to transfer (max is 1024 words)
// Outputs: none
// This routine does not wait for completion
void DMA_Xfr(uint32_t *src, uint32_t *dest, uint32_t cnt){
    ucControlTable[CH30] = (uint32_t)src+cnt*4-1; // last address
    ucControlTable[CH30+1] = (uint32_t)dest+cnt*4-1; // last address
    ucControlTable[CH30+2] = 0xAA00C002+((cnt-1)<<4); // Control Word
/* DMACHCTL      Bits  Value Description
DSTINC          31:30  2   32-bit destination address increment
DSTSIZE         29:28  2   32-bit destination data size
SRCINC          27:26  2   32-bit source address increment
SRCSIZE         25:24  2   32-bit source data size
reserved       23:18  0   Reserved
ARBSIZE         17:14  3   Arbitrates after 8 transfers
XFERSIZE        13:4  cnt-1 Transfer cnt items
NXTUSEBURST     3      0   N/A for this transfer type
XFERMODE        2:0    2   Use Auto-request transfer mode
*/
UDMA_ENASET_R = BIT30; // µDMA Channel 30 is enabled.
UDMA_SWREQ_R = BIT30; // software start,
}

```

*Program 7.1. Memory to memory transfer using DMA (DMASoftware\_4C123).*

In this next example, the user initializes the SPI port, initializes the DMA, enables interrupts and starts the DMA transfer by passing a pointer to a data array. The array **SinTable** contains a 256-entry 12-bit sine wave. The data must be stored in RAM, because we cannot use DMA to transfer to or from ROM. The main program is shown in Program 7.2.

```

uint16_t SinTable[256] = {
2048,2097,2146,2195,2244,2293,2341,2390,2438,2486,2534,2581,2629,2675,2722,2768,
2813,2858,2903,2947,2991,3034,3076,3118,3159,3200,3239,3278,3317,3354,3391,3427,
3462,3496,3530,3562,3594,3625,3654,3683,3711,3738,3763,3788,3812,3834,3856,3876,
3896,3914,3931,3947,3962,3976,3988,3999,4010,4019,4026,4033,4038,4043,4046,4047,

```



```

4048,4047,4046,4043,4038,4033,4026,4019,4010,3999,3988,3976,3962,3947,3931,3914,
3896,3876,3856,3834,3812,3788,3763,3738,3711,3683,3654,3625,3594,3562,3530,3496,
3462,3427,3391,3354,3317,3278,3239,3200,3159,3118,3076,3034,2991,2947,2903,2858,
2813,2768,2722,2675,2629,2581,2534,2486,2438,2390,2341,2293,2244,2195,2146,2097,
2048,1999,1950,1901,1852,1803,1755,1706,1658,1610,1562,1515,1467,1421,1374,1328,
1283,1238,1193,1149,1105,1062,1020,978,937,896,857,818,779,742,705,669,634,600,
566,534,502,471,442,413,385,358,333,308,284,262,240,220,200,182,165,149,134,120,
108,97,86,77,70,63,58,53,50,49,48,49,50,53,58,63,70,77,86,97,108,120,134,149,165,
182,200,220,240,262,284,308,333,358,385,413,442,471,502,534,566,600,634,669,705,
742,779,818,857,896,937,978,1020,1062,1105,1149,1193,1238,1283,1328,1374,1421,
1467,1515,1562,1610,1658,1706,1755,1803,1852,1901,1950,1999};

```

```

int main(void){
    PLL_Init();           // now running at 80 MHz
    DAC_Init(0x1000);    // initialize with command: Vout = Vref
    DMA_Init(625);       // DMA channel 8 for Timer5A, every 7.8125us
    EnableInterrupts(); // Timer5A interrupt on completion, every 2ms
    DMA_Start(SinTable, SSI0_DR, 256); //7.8125us*256= 2ms period sine wave
    while(1){
    }
}

```

*Program 7.2. Main program to create a continuous sin wave using DMA (DMASPI\_4C123).*

The low-level driver is presented in Program 7.3. Every 7.8125  $\mu$ s 16 bits from the SinTable are copied from RAM to the SSI0 data register. This is a cycle-steal DMA with one bus cycle used to read from the SinTable and a second bus cycle to write to SSI0\_DR . After 256 transfers, which will be every 2 ms, a Timer 5 interrupt occurs, and the process continues using ping-pong mode. As long as the interrupt Timer 5 ISR is run within 2 ms of its trigger, this system is real-time with virtually no output jitter. DMA requests can occur in the middle of instructions, and will occur regardless of processor state. The only events that can stall a DMA are another higher priority DMA requests.

```

// The control table used by the uDMA controller.
uint32_t ucControlTable[256] __attribute__((aligned(1024)));
// Timer5A uses uDMA channel 8 encoding 3
// channel 8 is at indices 32, 33, 34 (primary source,destination,control) and
//          at indices 160,161,162 (alternate source,destination,control)
#define CH8 (8*4)
#define CH8ALT (8*4+128)
#define BIT8 0x00000100
// ***** Timer5A_Init *****
// Activate Timer5A trigger DMA periodically
// Inputs: period in 12.5nsec
// Outputs: none
void Timer5A_Init(uint16_t period){ volatile uint32_t Delay;
    SYSCCTL_RCGCTIMER_R |= 0x20; // 0) activate timer5
    Delay = 0; // wait for completion
    TIMER5_CTL_R &= ~0x00000001; // 1) disable timer5A during setup
    TIMER5_CFG_R = 0x00000004; // 2) configure for 16-bit timer mode
    TIMER5_TAMR_R = 0x00000002; // 3) configure for periodic mode,
    TIMER5_TAILR_R = period-1; // 4) reload value
    TIMER5_TAPR_R = 0; // 5) 12.5ns timer5A
    TIMER5_ICR_R = 0x00000001; // 6) clear timer5A time out flag
    TIMER5_IMR_R |= 0x00000001; // 7) arm time out interrupt
    NVIC_PRI23_R = (NVIC_PRI23_R&0xFFFFF00)|0x00000040; // 8) priority 2

```

```

// interrupts enabled in the main program after all devices initialized
// vector number 108, interrupt number 92
}
// *****DMA_Init*****
// Initialize the buffer to port transfer, triggered by timer 5A
// This needs to be called once before requesting a transfer
// The source address increments by 2, destination address is fixed
// Call DMA_Start to begin continuous transfer, call DMA_Stop to halt
// Inputs: period in 12.5nsec Outputs: none
void DMA_Init(uint16_t period){int i; volatile uint32_t delay;
    for(i=0; i<256; i++){
        ucControlTable[i] = 0;
    }
    SYSTCTL_RCGCDMA_R = 0x01; // μDMA Module Run Mode Clock Gating Control
    delay = SYSTCTL_RCGCDMA_R; // allow time to finish
    UDMA_CFG_R = 0x01; // MASTEN Controller Master Enable
    UDMA_CTLBASE_R = (uint32_t)ucControlTable;
    UDMA_CHMAP1_R = (UDMA_CHMAP1_R&0xFFFFFFF0)|0x00000003; // timer5A
    UDMA_PRIOLR_R = BIT8; // default, not high priority
    UDMA_ALTCLR_R = BIT8; // use primary control
    UDMA_USEBURSTCLR_R = BIT8; // responds to both burst and single requests
    UDMA_REQMASKCLR_R = BIT8; // allow the μDMA controller to recognize requests
    Timer5A_Init(period);
}
uint16_t *SourcePt; // last address of source buffer, inc by 2
volatile uint32_t *DestinationPt; // fixed address
uint32_t Count; // number of halfwords to transmit
// private function used to reprogram regular channel control structure
void static setRegular(void){
    ucControlTable[CH8] = (uint32_t)SourcePt; // first and last address
    ucControlTable[CH8+1] = (uint32_t)DestinationPt; // last address
    ucControlTable[CH8+2] = 0xD5000003+((Count-1)<<4); // DMA Channel Control Word
/* DMACHCTL Bits Value Description
DSTINC 31:30 11 no destination address increment
DSTSIZE 29:28 01 16-bit destination data size
SRCINC 27:26 01 16-bit source address increment, +2
SRCSIZE 25:24 01 16-bit source data size
reserved 23:18 0 Reserved
ARBSIZE 17:14 0 Arbitrates after 1 transfer
XFERSIZE 13:4 count-1 Transfer count items
NXTUSEBURST 3 0 N/A for this transfer type
XFERMODE 2:0 011 Use ping-pong transfer mode */
}
// private function used to reprogram alternate channel control structure
void static setAlternate(void){ // same as regular
    ucControlTable[CH8ALT] = (uint32_t)SourcePt; // first and last address
    ucControlTable[CH8ALT+1] = (uint32_t)DestinationPt; // last address
    ucControlTable[CH8ALT+2] = 0xD5000003+((Count-1)<<4); // DMA Channel Control
}
// *****DMA_Start*****
// Called to transfer halfwords from source to destination
// The source address is incremented by two each 16-bit xfer, destination fixed
// Inputs: source is a pointer to a RAM buffer containing waveform to output
// destination is a pointer to 32-bit DAC device (SSI0_DR_R),
// count is the number of halfwords to transfer (max is 1024 halfwords)
// Outputs: none
// This routine does not wait for completion, runs continuously
void DMA_Start(uint16_t *source, volatile uint32_t *destination, uint32_t count){
    SourcePt = source+count-1; // last address of source buffer

```

```

DestinationPt = destination;
Count = count;          // number of halfwords to transmit
setRegular();
setAlternate();
NVIC_EN2_R = 0x10000000;    // 9) enable interrupt 92 in NVIC
// vector number 108, interrupt number 92
TIMER5_CTL_R |= 0x00000001; // 10) enable timer5A
UDMA_ENASET_R |= BIT8; //  $\mu$ DMA Channel 8 is enabled
// bits 2:0 ucControlTable[CH8+2] become clear when regular structure done
// bits 2:0 ucControlTable[CH8ALT+2] become clear when alternate structure done
}
uint32_t NumberOfBuffersSent=0;
// *****DMA_Status*****
// Can be used to check the status of the continuous DMA transfer
// Inputs: none
// Outputs: the number of buffers transferred
uint32_t DMA_Status(void){
    return NumberOfBuffersSent;
}
void Timer5A_Handler(void){ // interrupts after each block is transferred
    TIMER5_ICR_R = TIMER_ICR_TATOCINT; // acknowledge timer5A timeout
    NumberOfBuffersSent++;
    if((ucControlTable[CH8+2]&0x0007)==0){ // regular buffer complete
        setRegular(); // rebuild channel control structure
    }
    if((ucControlTable[CH8ALT+2]&0x0007)==0){ // Alternate buffer complete
        setAlternate(); // rebuild channel control structure
    }
}
// *****DMA_Stop*****
// Stop the transfer halfwords from source to destination
// Inputs: none      Outputs: none
void DMA_Stop(void){
    UDMA_ENACL_R = BIT8; //  $\mu$ DMA Channel 8 is disabled
    NVIC_DIS2_R = 0x10000000; // 9) disable interrupt 92 in NVIC
    TIMER5_CTL_R &= ~0x00000001; // 10) disable timer5A
}

```

*Program 7.3. Memory to DAC transfer using DMA (DMASPI\_4C123).*

---

## 7.6. Exercises

7.1 For each term give a definition in 32 words or less.

- a) Latency
- b) Real-time
- c) DMA
- d) Seek time
- e) Bandwidth
- f) Dual-port memory
- g) Bank-switched memory
- h) Double buffer

7.2 For each pair of terms, explain the similarities and differences in 32 words or less

- a) Burst versus cycle-steal DMA
- b) Single address versus dual address DMA
- c) Back buffer versus front buffer
- d) Write data required versus write data available

7.3 The objective of this problem is to interface various devices to the computer using DMA synchronization. You may assume the bus bandwidth is at least 8 million bytes/sec. For each device you are asked to select the most appropriate DMA mode. Assume the devices support single address DMA. The 16-bit address of the memory buffer used in each case is 0x1234. Fill in the table with the most appropriate mode for each Device

**Write Tape Drive** Each tape block is 256 bytes. When a tape head is ready, the controller will signal that it is ready to accept all 256 bytes. At this time, the tape interface chip is ready to transfer as fast as possible all 256 bytes from the memory buffer at 0x1234 to the tape.

**Sound Input** The sound waveform buffer is located in memory at 0x1234. Your interface will read the 8-bit ADC 1024 times at 22 kHz and store the data in the buffer. Your software will be smart enough to create two 512 byte buffers out of the 1024 bytes (double buffer) so that it can process one buffer while the ADC data is being stored automatically under DMA control into the other buffer. I.e., when the 1024 byte wave buffer has been filled, the DMA system should repeat and fill it up again.

**Read Hard Drive** There is a 256-byte buffer at 0x1234 that your DMA system will fill with data from the hard disk. When a hard drive read head is ready, the controller will signal that it has the next byte from the disk. It takes 10ms for the read head to be ready, then the 256 bytes of data can be transferred from the disk to memory at 2 million bytes/sec.

	Tape	Sound	Disk
--	------	-------	------

Cycle Steal or Block Transfer			
Read or Write Transfer			
Autoinitialization (Yes or No)			
Address increment or decrement			
DMA Address register value			
DMA Count register value			

**7.4** When a 256-byte block is written to a floppy disk, there are 256 separate single-address DMA cycles in cycle steal mode. This question deals with just one of these DMA transfers. There are 14 events listed below. First you will eliminate the events that do not occur during the DMA cycle that saves one byte on the disk. In particular, list the events that will not occur. Second, you will list the events that do occur in the proper sequence.

- a) An interrupt is requested.
- b) Registers are pulled from the stack.
- c) Registers are pushed on the stack.
- d) The DMAC asks the processor to halt by activating its **Halt** signal.
- e) The DMAC deactivates its **Halt** request to the processor.
- f) The DMAC tells the FDC interface that a DMA cycle is occurring by activating its **Ack** signal; the DMA Controller drives the address bus with the FDC address; the DMAC drives the control bus to signify a write cycle (e.g., R/W=0); the memory drives the data bus; the FDC accepts the data.
- g) The DMAC tells the FDC interface that a DMA cycle is occurring by activating its **Ack** signal; the DMAC drives the address bus with the memory address; the DMAC drives the control bus to signify a memory read cycle (e.g., R/W=1); the memory drives the data bus; the FDC accepts the data.
- h) The FDC deactivates its DMA **Request** signal to the DMAC.
- i) The FDC requests a DMA cycle to the DMAC by activating its **Request** signal.
- j) The interrupt service routine is executed.
- k) The write head is properly positioned over the place on the disk.
- l) The processor address and control lines float; the processor responds to the DMAC that it is halted by activating its **HaltAck** signal.
- m) The processor resumes software execution.
- n) Wait until the current instruction is finished executing.

# 8. File system management

## Chapter 8 objectives are to:

- Present the fundamentals of file system management
- Develop a detailed solution of a simple file system
- Define basic components of a FAT system
- Describe how to program internal flash memory
- Present interfacing methods to a secure digital card (SDC)

In this chapter, we present approaches for managing large amounts of data on an embedded system. We present two methods to save and retrieve data: internal flash and an external secure digital card. In particular, we will define data as abstract elements (files) and then create a mapping from the logical to the physical. We will present methods for creating directory, accessing data, and managing free space.

We will begin this chapter with an introduction of file systems. In particular, we briefly present what is a file system, discuss how it will be used, develop performance metrics, present fundamental concepts, and then conclude with a couple of simple examples.

Embedded applications that might require disk storage include data acquisition, database systems, and signal generation systems. You can also use a disk in an embedded system to log debugging information.

---

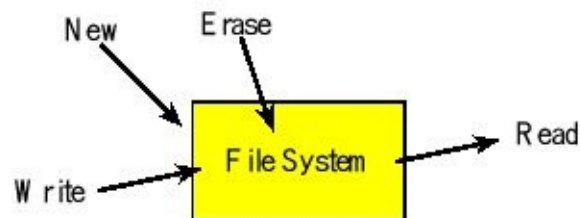
# 8.1. Performance Metrics

## 8.1.1. Usage

A file system allows the software to store data and to retrieve previously stored data, see Figure 8.1. Typically, the size of the stored data exceeds available memory of the computer. In general, file systems allow for these operations:

- Create a new file
- Write data to the file (append to end or insert at arbitrary location)
- Read data from the file (read sequential or read at arbitrary location)
- Erase the file

Each file will have a name or a number, with which we will use to access the data in that file. In general, we can organize files into directories. However, in this chapter, we will restrict our file system implementations to a single directory containing all files.



*Figure 8.1. A file system is used to store data.*

When designing a file system, it is important to know how it will be used. We must know if files will be erased. In particular, we can simplify how the disk is organized if we know files, once created, will never be destroyed.

For example, when recording and playing back sound and images, the data will be written and read in a sequential manner. We call this use pattern as **sequential access**. If we are logging or recording data, then we will need to append data at the end of a file but never change any data once logged. Conversely, an editor produces more of a **random access** pattern for data reading and writing. Furthermore, an editor requires data insertion and removal anywhere within the file. If the file is used as a data base, then the positions in the file where we read will be random (random access reading). However, the data base may be static, in other words, it may only need to be written once.

The **reliability** of the storage medium and the cost of lost information will also affect the design of a file system. For an embedded system we can improve reliability by

selecting a more reliable storage medium or by deploying redundancy. For example, we could write the same data into three independent disks, and when reading we read all three and return the median of the three data values.

So in general, we should first study the use cases in our system before choosing or designing the file system. In this chapter, we will develop in detail a file system for data logging, where both writing and reading will be done sequentially, and files will never be deleted.

## 8.1.2. Specifications

There are many organizational approaches when designing a file system. As we make design decisions, it is appropriate to consider both quantitative and qualitative parameters. We can measure the effectiveness of a file system by

- Maximum file size
- Maximum number of files
- Speed to read data at a random position in the file
- Speed to read data in a sequential fashion
- Speed to write data into the file

## 8.1.3. Fragmentation

**Internal fragmentation** is storage that is allocated for the convenience of the operating system but contains no information. This space is wasted. Often this space is wasted in order to improve speed or to provide for a simpler implementation. The fragmentation is called "internal" because the wasted storage is inside the allocated region, see Figure 8.2. In most file systems, whole sectors (or even clusters of sectors) are allocated to individual files, because this simplifies organization and makes it easier to grow files. Any space left over between the last byte of the file and the first byte of the next sector is a form of internal fragmentation called **file slack** or **slack space**. A small file holding  $m$  bytes is allocated an entire sector capable of storing  $n$  bytes of data. However, only  $m$  of those locations contains data, so the remaining  $n-m$  bytes can be considered internal fragmentation. The pointers and counters used by the OS to manage the file are not considered internal fragmentation, because even though the locations do not contain data, the space is not wasted. Whether or not to count the OS pointers and counters as internal fragmentation is a matter of debate. As is the case with most definitions, it is appropriate to document your working definition of internal fragmentation whenever presenting performance specifications to your customers.



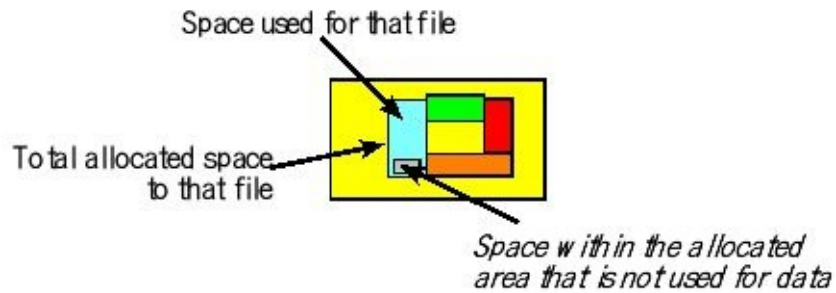


Figure 8.2. The large block is the entire disk. There are multiple files (rectangles) on this disk. The rectangle on the left represents one file. Within the allocated space for this file there is data, and there is some space in the allocated area that is not data. The space within the allocated area not used for data is internal fragmentation.

Many compilers will align variables on a 32-bit boundary, even though memory is byte-addressable. If the size of a data structure is not divisible by 32 bits, it will skip memory bytes so the next variable is aligned onto a 32-bit boundary. This wasted space is also internal fragmentation.

**Checkpoint 8.1:** If the sector size is  $n$  and the size of the files is randomly distributed, what is the average internal fragmentation per file?

**External fragmentation** exists when the largest file that can be allocated is less than the total amount of free space on the disk. External fragmentation occurs in systems that require contiguous allocation, like a memory manager. External fragmentation would occur within a file system that allocates disk space in contiguous sectors. Over time, free storage becomes divided into many small pieces, see Figure 8.3. It is a particular problem when an application allocates and deallocates regions of storage of varying sizes. The result is that, although free storage is available, it is effectively unusable because it is divided into pieces that are too small to satisfy the demands of the application. The term "external" refers to the fact that the unusable storage is outside the allocated regions.

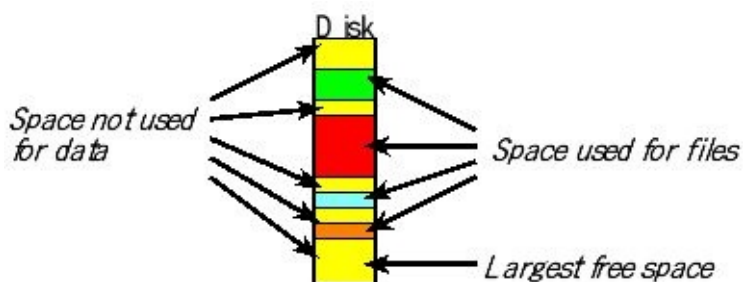


Figure 8.3. There are four files on this disk, and there are five sections of free space. The largest free space is less than the total free space, which is defined as external fragmentation, assuming the file system requires contiguous allocation.

For example, assume we have a file system employing contiguous allocation. A new file with five sectors might be requested, but the largest contiguous chunk of free disk space is only three sectors long. Even if there are ten free sectors, those free sectors may be separated by allocated files, one still cannot allocate the requested file with five sectors, and the allocation request will fail. This is external fragmentation because there are ten free sectors but the largest file that can be allocated is three sectors.

**Checkpoint 8.2:** Consider this analogy. You are given a piece of wood that is 10 meters long, and you are asked to cut it because you need one piece that is 2 meters long. What is the best way to cut the wood so there is no external fragmentation? Think of another way the wood could have been cut so the largest piece of free wood is smaller than the total free wood, creating external fragmentation?

---

## 8.2. File System Allocation

There are three components of the **file system**: the directory, allocation, and free-space management. This section introduces fundamental concepts and the next two sections present simple file systems. In this chapter, we define **sector** as a unit of storage. Whole sectors will be allocated to a file. In other words, we will not combine data from multiple files into a single sector.

We consider information in a file as a simple linear array of bytes. The “logical” address is considered as the index into this array. However, data must be placed at a “physical” location on the disk. An important task of the file system is to translate the logical address to the physical address (Figure 8.4).



*Figure 8.4. A file system must translate from a logical address to the physical address.*

### 8.2.1. Contiguous allocation

**Contiguous allocation** places the data for each file at consecutive sectors on the disk, as shown in Figure 8.5. Each directory entry contains the file name, the sector number of the first sector, the length in sectors. This method has similar theory as a memory manager. You could choose first-fit, best-fit, or worst-fit algorithms to manage storage. **First fit** is an algorithm that searches the available free space and selects the first area it fits that is large enough for the file needs. This algorithm executes quickly. **Best fit** is an algorithm that looks at all available free space and chooses the smallest area that is large enough for the file needs. Best-fit may limit external fragmentation for contiguous allocation schemes. **Worst fit** is an algorithm that looks at all available free space and chooses the largest area, assuming that area is large enough for the file needs.

If the file can increase in size, either you can leave no extra space, and copy the file elsewhere if it expands, or you can leave extra space when creating a new file. Assuming the directory is in memory, it takes only one disk sector read to access any data in the file. A disadvantage of this method is you need to know the maximum file size when a file is created, and it will be difficult to grow the file size beyond its initial allocation.

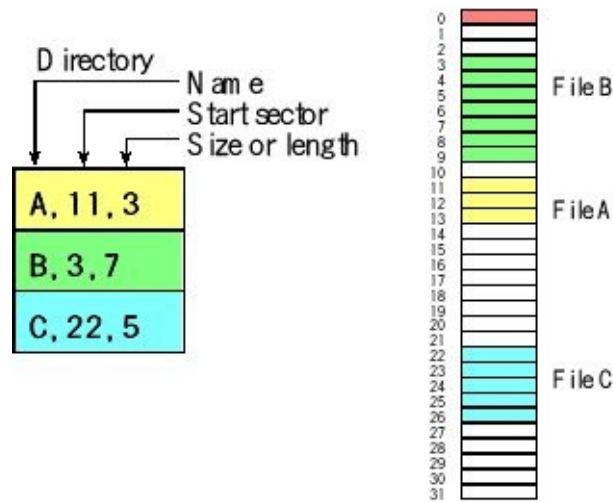


Figure 8.5. A simple file system with contiguous allocation. Notice all the sectors of a file are physically next to each other.

**Checkpoint 8.3:** The disk in Figure 8.5 has 32 sectors with the directory occupying sector 0. The disk sector size is 512 bytes. What is the largest new file that can be created?

**Checkpoint 8.4:** You wish to allocate a new file requiring 1 sector on the disk in Figure 8.5. Using first-fit allocation, where would you put the file? Using best-fit allocation, where would you put the file? Using worst-fit allocation, where would you put the file?

One of the tasks the file system must manage is free space. One simple scheme for free space management is a **bit table**. If the disk has  $n$  sectors, then we will maintain a table with  $n$  bits, assigning one bit for each sector. If the bit is 1, the corresponding sector is free, and if the bit is 0, the sector is used. Figure 8.5 shows a simple disk with 32 sectors. For this disk we could manage free space with one 32-bit number.

**Checkpoint 8.5:** Assume the sector size is 4096 bytes and the disk is one gibibyte. How many bytes would it take to maintain a bit table for the free space?

## 8.2.2. Linked allocation

**Linked allocation** places a sector pointer in each data sector containing the address of the next sector in the file, as shown in Figure 8.6. Each directory entry contains a file name and the sector number of the first sector. There needs to be a way to tell the end of a file. The directory could contain the file size, each sector could have a counter, or there could be an end-of-file marker in the data itself. Sometimes, there is also a pointer to the last sector, making it faster to add to the end of the file. Assuming the directory is in memory and the file is stored in  $N$  sectors, it takes on average  $N/2$  disk-sector reads to access any random piece of data on the disk. Sequential reading and writing are efficient, and it also will be easy to append data at the end of the file. Linked allocation has no external fragmentation.

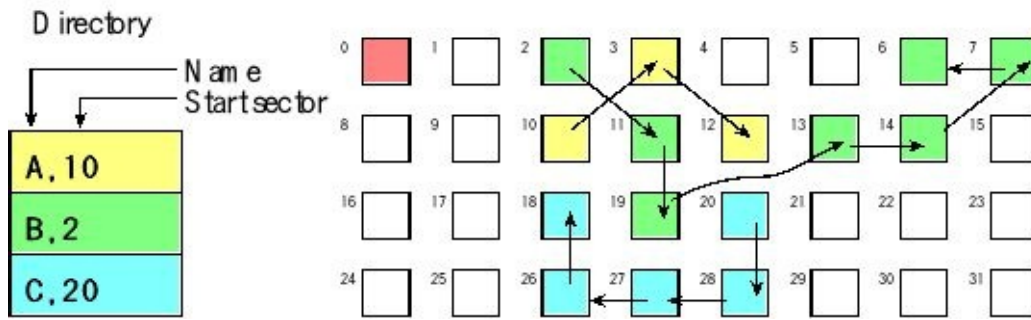


Figure 8.6. A simple file system with linked allocation.

**Checkpoint 8.6:** If the disk holds 2 Gibibytes of data broken into 512-byte sectors, how many bits would it take to store the sector address?

**Checkpoint 8.7:** If the disk holds 2 Gibibytes of data broken into 32k-byte sectors, how many bits would it take to store the sector address?

**Checkpoint 8.8:** The disk in Figure 8.6 has 32 sectors with the directory occupying sector 0. The disk-sector size is 512 bytes. What is the largest new file that can be created? Is there external fragmentation?

**Checkpoint 8.9:** How would you handle the situation where the number of bytes stored in a file is not an integer multiple of the number of data bytes that can be stored in each sector?

We can also use the links to manage the free space, as shown in Figure 8.7. If the directory were lost, then all file information except the filenames could be recovered. Putting the number of the last sector into the directory with double-linked pointers improves recoverability. If one data sector were damaged, then remaining data sectors could be rechained, limiting the loss of information to the one damaged sector.

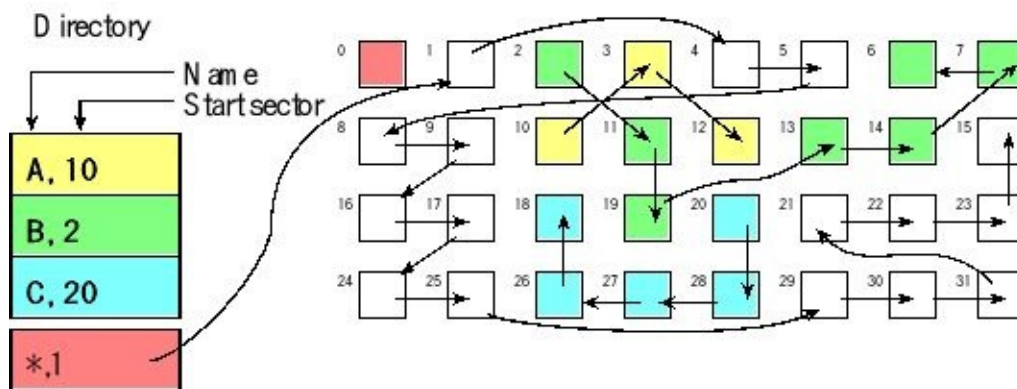


Figure 8.7. A simple file system with linked allocation and free space management.

### 8.2.3. Indexed allocation

**Indexed allocation** uses an index table to keep track of which sectors are assigned to

which files. Each directory entry contains a file name, an index for the first sector, and the total number of sectors, as shown in Figure 8.8. One implementation of indexed allocation places all pointers for all files on the disk together in one index table. Another implementation allocates a separate index table for each file. Often, this table is so large it is stored in several disk sectors. For example, if the sector number is a 16-bit number and the disk sector size is 512 bytes, then only 256 index values can be stored in one sector. Also for reliability, we can store multiple copies of the index on the disk. Typically, the entire index table is loaded into memory while the disk is in use. The RAM version of the table is stored onto the disk periodically and when the system is shut down. Indexed allocation is faster than linked allocation if we employ random access. If the index table is in RAM, then any data within the file can be found with just one sector read. One way to improve reliability is to employ both indexed and linked allocation. The indexed scheme is used for fast access, and the links can be used to rebuild the file structure after a disk failure. Indexed allocation has no external fragmentation.

**Checkpoint 8.10:** If the sector number is a 16-bit number and the sector size is 512 bytes, what is the maximum disk size?

**Checkpoint 8.11:** A disk with indexed allocation has 2 GiB of storage. Each file has a separate index table, and that index occupies just one sector. The disk sector size is 1024 bytes. What is the largest file that can be created? Give two ways to change the file system to support larger files.

**Checkpoint 8.12:** This disk in Figure 8.8 has 32 sectors with the directory occupying sector 0 and the index table in sector 1. The disk-sector size is 512 bytes. What is the largest new file that can be created? Is there external fragmentation?

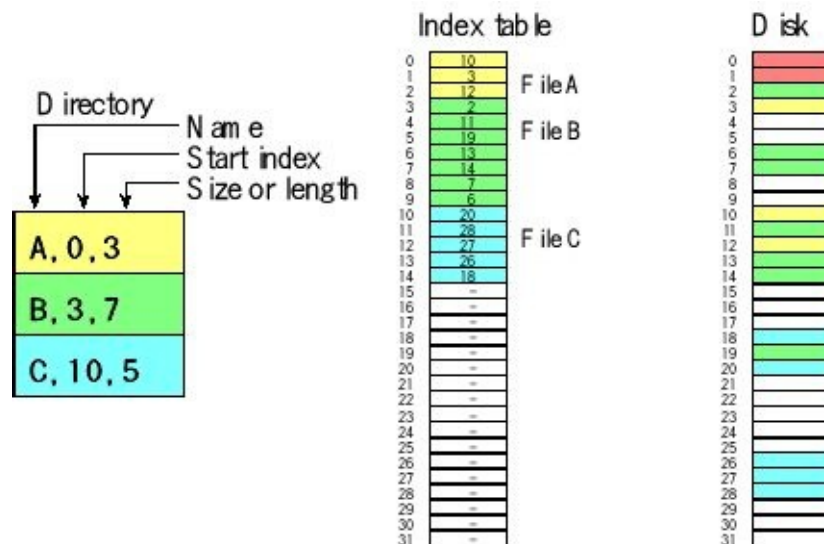


Figure 8.8. A simple file system with indexed allocation.



## 8.2.4. File allocation table (FAT)

The **file allocation table** (FAT) is a mixture of indexed and linked allocation, as shown in Figure 8.9. Each directory entry contains a file name and the sector number of the first sector.

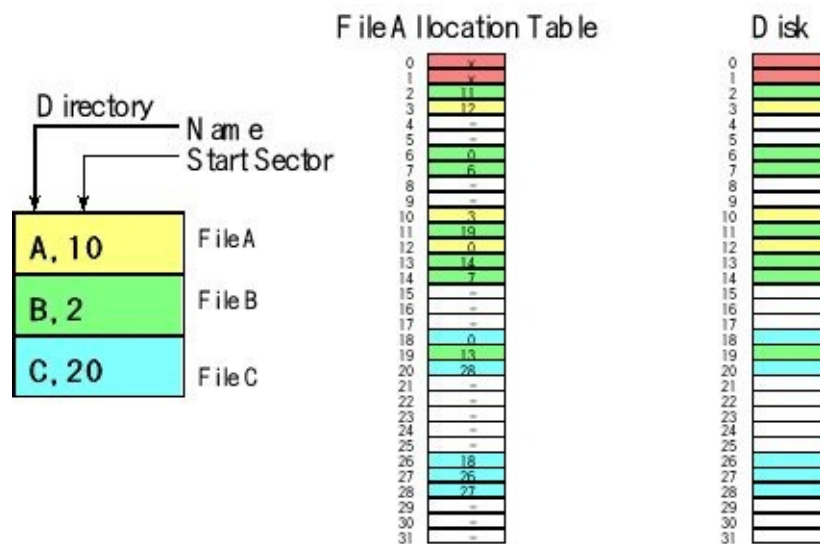


Figure 8.9. A simple file system with a file allocation table.

The FAT is just a table containing a linked list of sectors for each file. Figure 8.9 shows file A in sectors 10, 3, and 12. The directory has sector 10, which is the initial sector. The FAT contents at index 10 is a 3, so 3 is the second sector. The FAT contents at index 3 is a 12, so 12 is the third sector. The FAT contents at index 12 is a NULL, which means there are no more sectors in the file. A FAT allocation schemes have no external fragmentation.

Many scientists classify FAT as a “linked” scheme, because it has links. However, other scientists call it an “indexed” scheme, because it has the speed advantage of an “indexed” scheme when the table for the entire disk is kept in main memory. If the directory and FAT are in memory, it takes just one disk read to access any data in a file. If the disk is very large, the FAT may be too large to fit in main memory. If the FAT is stored on the disk, then it will take 2 or 3 disk accesses to find an element within the file. The - in Figure 8.9 represent free sectors. In Figure 8.10, we can chain them together in the FAT to manage free space.

**Checkpoint 8.13:** This disk in Figure 8.10 has 32 sectors with the directory occupying sector 0 and the FAT in sector 1. The disk sector size is 512 bytes. What is the largest new file that can be created? Is there an external fragmentation?

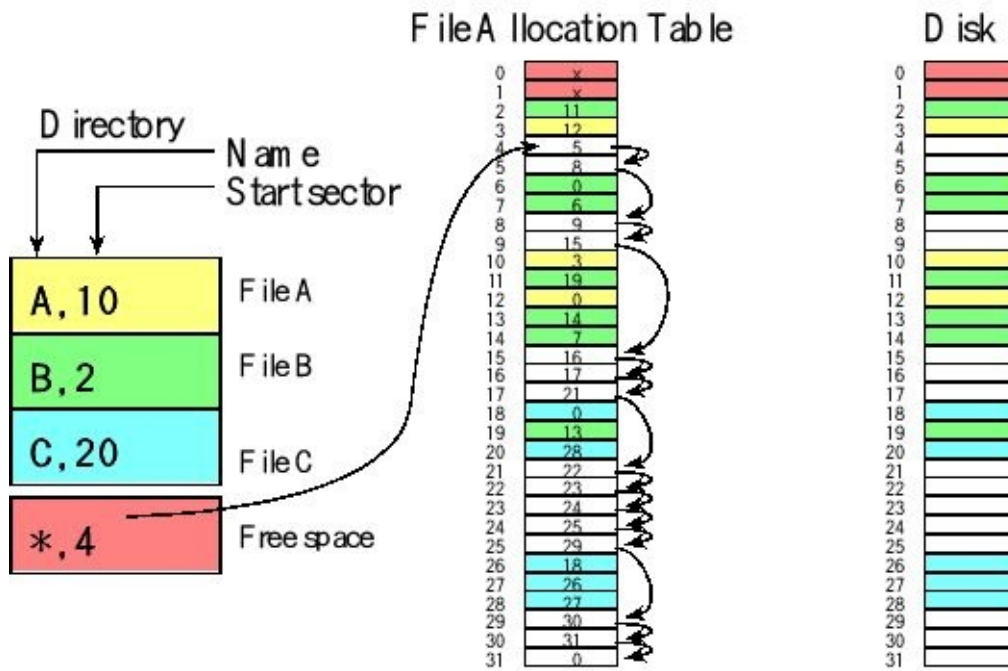


Figure 8.10. The simple file system with a file allocation table showing the free-space management.

**Observation:** In this section we use 0 to mean null pointer. Later in the chapter we will use 255 to mean null pointer. We use 0 in this section because this discussion is similar to the standard FAT16. However, for EEPROM-based systems, we need to use 255 because 255 is the value that occurs when the flash memory is erased.



---

## 8.3. Solid State Disk

### 8.3.1. Flash memory

In general, we can divide memory into volatile and nonvolatile categories. Volatile means it loses its data when power is removed and restored. Nonvolatile means it retains its data when power is removed and restored. There are many types of memory, but here are four of them

- Volatile memory
  - Static random access memory, SRAM
  - Dynamic random access memory, DRAM
- Nonvolatile memory
  - Flash electrically erasable programmable read only memory, EEPROM
  - Ferroelectric random access memory, FRAM

As you know data and the stack are allocated in RAM, because it needs read/write access. DRAM has fewer transistors/bit compared to SRAM because it does require periodic refreshing. Most Cortex M microcontrollers use SRAM because of its simple technology and ability to operate on a wide range of bus frequencies. For random access memories, there is a size above which DRAM is more cost effective than SRAM. Dynamic random access memory (DRAM) is the type of memory found in most personal computers. Embedded devices like the Beaglebone and Raspberry Pi also use DRAM.

**Ferroelectric RAM (FRAM)** is a random access memory similar to DRAM but uses a ferroelectric layer instead of a dielectric layer. The ferroelectric layer provides the non-volatility needed for program storage. Some new lines of microcontrollers use FRAM instead of flash EEPROM for their non-volatile storage. The MSP430FRxx microcontrollers from Texas Instruments use FRAM to store programs and data in one shared memory object. Other companies that produce FRAM microcontrollers include Fujitsu and Silicon Labs. FRAM requires less power usage, has a faster write, and provides a greater maximum number of write-erase cycles when compared to flash. When compared to flash, FRAMs have lower storage densities, smaller sizes, and higher cost.

**Solid-state disks** can be made from any nonvolatile memory, such as battery-backed RAM, FRAM, or flash EEPROM. Personal computers typically use disks made with magnetic storage media and moving parts. While this magnetic-media technology is acceptable for the personal computer because of its large storage size (> 1 Tibibyte)

and low cost (<\$100 OEM), it is not appropriate for an embedded system because of its physical dimensions, electrical power requirements, noise, sensitivity to motion, and weight.

**Secure digital (SD)** cards use Flash EEPROM together with interface logic to read and write data. For an embedded system we could create a file system using an SD card or using the internal flash of the microcontroller itself. SD cards are an effective approach when file storage needs exceed 128 kibibytes, because of the low cost and simple synchronous serial interface. If we use the internal flash of the microcontroller itself, there will be no additional costs to developing this file system.

Smart phones, tablets, and cameras currently employ solid-state disks because of their small physical size and low power requirements. Unfortunately, solid-state disks have smaller storage sizes and higher cost/bit than the traditional magnetic storage disk. A typical 64-Gibibyte SD card costs less than \$20. The cost/bit is therefore about \$300/Tibibyte. In contrast, an 8-Tibibyte hard drive costs about \$200 or \$25/Tibibyte. The cost/bit of flash storage is expensive as compared to a traditional hard drive. However, there is a size point (e.g., below 128 Gibibyte), below which the overall cost of flash will be less than a traditional magnetic/motorized drive.

A **flash memory cell** uses two transistors; the gates of the two transistors are positioned gate to gate separated by an insulation layer as shown in Figure 8.11. Because each flash bit has only two transistors, the microcontroller can pack more flash bits into the chip as compared to SRAM or FRAM bits. A normal transistor has an input gate that is used to control conductance between the source and drain. However, in a flash memory cell, one of the gates is floating, which means it is not connected to anything. If we trap charge on this floating gate, we define this state as value 0. If there is no trapped charge, we define the state as a 1. There are three operations we can perform on the cell.

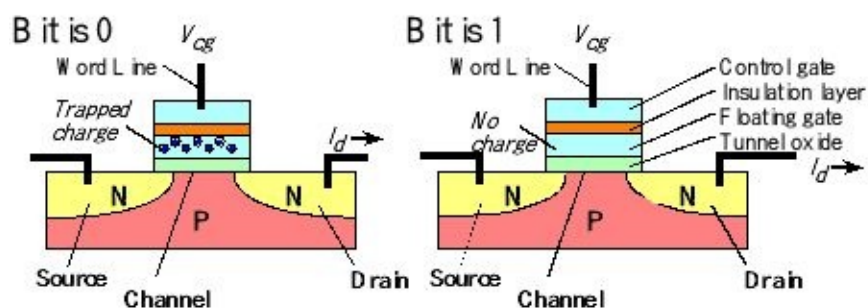


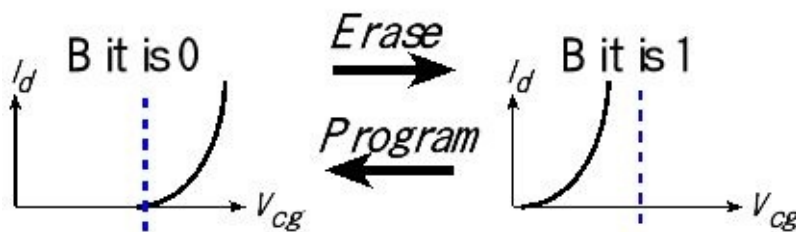
Figure 8.11. The floating gate in a flash memory cell creates the storage.

If we place a large voltage on the control gate ( $V_{cg}$ ), we can get all the trapped charge to flow from the floating gate to the source below, hence **erasing** the cell, making its value equal to 1.

Conversely if we place a large voltage of the opposite polarity on the control gate,

we can add charge to the floating gate, **programming** its value equal to 0. On the TM4C123 the smallest granularity with which we can erase is 1024 bytes. On the MSP432 we erase flash in blocks of 4096 bytes. However, we can program individual words on most flash memories including the TM4C123 and MSP432. Once erased to a 1 or programmed to a 0, the charge or lack of charge remains on the floating gate even if power is removed from the system. Hence, this memory is nonvolatile. Data in the TM4C123 and MSP432 flash memories will remain valid for 20 years, and the memory will operate up to 100,000 erase/program cycles. Erasing and programming operations take a very long time compared to writing static RAM (SRAM). For example, it takes 8 to 15 ms to erase an entire 1024-byte page on the TM4C123. In contrast, writing 256 words in RAM on an 80-MHz Cortex-M takes 5 cycles/loop, which adds up to 1280 cycles or 16  $\mu$ s.

To **read** the value from flash, the control gate is activated. There is a threshold voltage for the control gate at which source-drain current ( $I_d$ ) flows if the bit is 0 and will not flow if the bit is 1. The threshold voltage is depicted as the dotted line in Figure 8.12.



*Figure 8.12. The trapped charge in the floating gate affects the relationship between control gate voltage and drain current.*

For more information on flash see <http://computer.howstuffworks.com/flash-memory.htm>

For information on RAM memory see <http://computer.howstuffworks.com/ram.htm>

In summary:

- Flash memory cells have two transistors, so it has very high density
- Nonvolatile behavior implemented as trapped/no charge on the floating gate
- We can erase an entire block (1k or 4k), making all bits 1
- We can program individual bytes/words, making bits 0 as needed
- Both erasing and programming are very slow compared to reading

## 8.3.2. Flash device driver

One inexpensive approach to developing a file system is to use the internal flash storage of the microcontroller. Both the TM4C123 and MSP432 have 256 kibibytes of internal flash, existing from addresses 0 to 0x0003FFFF. Normally, we use the internal flash to save the machine code of our software. However, in this chapter we will allocate half of the flash, which is 128 kibibytes, to create a solid state disk. We divide the disk into **sectors** and operate on a sector by sector basis. Typically, the sector size is a power of 2; let each sector be  $2^p$  bytes. This means we will partition the  $2^{17}$ -byte disk into  $2^m$  sectors, where  $m+p=17$ . In general, there are three operations: we can erase (set bits to 1), program (set bits to 0), and read. The physical layer functions provide these basic operations. Program 8.1 shows the prototypes for the TM4C123. We do not need physical layer functions to read the flash, because once erased and programmed, software simply reads from the memory address in the usual manner. The TM4C123 is optimized for programming up to 128-byte (32-word) aligned “mass writes” or “fast writes”. The MSP432 implements this feature for up to 64-byte (16-word) arrays. The smallest block that we can erase on the TM4C123 is 1024 bytes. On the MSP432 we erase flash in blocks of 4096 bytes.

```
//-----Flash_Erase-----  
// Erase 1 KB block of flash on TM4C123, 4KB on MSP432  
// Input: addr 1-KB aligned flash memory address to erase  
// Output: 0 if successful, 1 if fail  
int Flash_Erase(uint32_t addr);  
  
//-----Flash_Write-----  
// Write 32-bit data to flash at given address.  
// Input: addr 4-byte aligned flash memory address to write  
//      data 32-bit data  
// Output: 0 if successful, 1 if fail  
int Flash_Write(uint32_t addr, uint32_t data);  
  
//-----Flash_WriteArray (TM4C123 only) -----  
// Write an array of 32-bit data to flash starting at given address.  
// Input: source pointer to array of 32-bit data  
//      addr 4-byte aligned flash memory address to start writing  
//      count number of 32-bit writes  
// Output: number of successful writes; return value == count if ok  
// Note: at 80 MHz, it takes 678 usec to write 10 words  
int Flash_WriteArray(uint32_t *source, uint32_t addr, uint16_t count);
```

*Program 8.1. Prototypes for the physical layer functions to manage the flash (4-k erase for MSP432).*

### 8.3.3. eDisk device driver

We will add an abstraction level above the physical layer to create an object that behaves like a disk. In particular, we will use 128 kibibytes of flash at addresses 0x00020000 to 0x0003FFFF to create the solid state disk and partition the disk into 512-byte sectors. This abstraction will allow us to modify the physical layer without modifying the file system code. For example, we might change the physical layer to a secure digital card, to a battery-backed RAM, to an FRAM, or even to network storage.

On most disks, there is physical partitioning of the storage into **blocks** in order to optimize for speed. For example, the smallest block on the MSP432 that we can erase is 4 kibibytes, and on the TM4C123 the block size is 1 kibibyte. We will use the term **block** to mean a physical partition created by the hardware, and use the term **sector** (which can be 1 or more blocks) as a logical partition defined by the operating system. In a file system, we will partition the disk into **sectors** and allocate whole sectors to a single file. In other words, we will not store data from two files into the same sector. This all or nothing allocation scheme is used by most file systems, because it simplifies implementation.

If we were to implement a file system that allows users to erase, move, insert (grow) or remove (shrink) data in the files, then we would need to erase blocks dynamically. Because the smallest block on the MSP432 that we can erase is 4096 bytes, we would have to choose a sector size that is an integer multiple of 4k. On the TM4C123 smallest sector size would be 1k. Unfortunately, a disk made from the 128k of the flash with 4k-sectors would only have 32 sectors. 32 is such a small number the file system would be quite constrained.

The philosophy of this book has been to implement the simplest system that still exposes the fundamental concepts. Therefore, in this chapter we will develop a simple file system that does not allow the user to delete, move, grow, or shrink data in the files. It does however allow users to create files and write data to a file in increments of sectors. More specifically, when writing we will always append data to the end of the file. We call this simple approach as a **write-once file system**. We will erase the 128k flash once, and then program 0's into the flash memory dynamically as it runs. Data logging and storage of debug information are applications of a write-once file system. For this simple file system, we can choose the sector size to be any size, because the flash is erased only once, and data is programmed as the user creates and writes sectors to the file. The size of the disk is 128 kibibytes, i.e.,  $2^{17}$  bytes. If the sector size is  $2^n$ , then there will be  $2^{17-n}$  sectors. For this system, if we were to use the fast write capabilities of the TM4C123 we could partition the 128 kibibyte disk as 1024 sectors with 128 bytes in each sector. Conversely, if we use the regular write function ( **Flash\_WriteArray** ) then we could choose any sector size. In Section 8.5, we will partition the disk into 256 sectors with 512 bytes per sector creating a file system where the sector address is an 8-bit number.

Program 8.2 shows the prototypes of the disk-level functions. **eDisk\_Init()** has no operations to perform in this system. It was added because other disks, like the SD card, will need initialization. You should have **eDisk\_Init** return zero if the drive parameter is 0 and return 1 if the drive parameter is not zero, because there is only one drive.

Reading a sector requires an address translation. The function **eDisk\_ReadSector** will copy 512 bytes from flash to RAM. The start of the disk is at flash address 0x00020000. Each sector is 512 bytes long, so the starting address of the sector is simply

$$0x00020000 + 512 * \text{sector}$$

Writing a sector requires the same address translation. The function **eDisk\_WriteSector** will program 512 bytes from RAM into flash. In particular, it will do the address translation and call the function **Flash\_WriteArray**. 512 bytes is 128 words, so the count parameter will be 128.

```

//***** eDisk_Init *****
// Initialize the interface between microcontroller and disk
// Inputs: drive number (only drive 0 is supported)
// Outputs: status
// RES_OK    0: Successful
// RES_ERROR 1: Drive not initialized
enum DRESULT eDisk_Init(uint32_t drive);

//***** eDisk_ReadSector *****
// Read 1 sector of 512 bytes from the disk, data goes to RAM
// Inputs: pointer to an empty RAM buffer
//        sector number of disk to read: 0,1,2,...255
// Outputs: result
// RES_OK    0: Successful
// RES_ERROR 1: R/W Error
// RES_WRPRT 2: Write Protected
// RES_NOTRDY 3: Not Ready
// RES_PARERR 4: Invalid Parameter
enum DRESULT eDisk_ReadSector(
    uint8_t *buff, // Pointer to a RAM buffer into which to store
    uint8_t sector); // sector number to read from

//***** eDisk_WriteSector *****
// Write 1 sector of 512 bytes of data to the disk, data comes from RAM
// Inputs: pointer to RAM buffer with information
//        sector number of disk to write: 0,1,2,...,255
```

```

// Outputs: result
// RES_OK    0: Successful
// RES_ERROR 1: R/W Error
// RES_WRPRT 2: Write Protected
// RES_NOTRDY 3: Not Ready
// RES_PARERR 4: Invalid Parameter
enum DRESULT eDisk_WriteSector(
    const uint8_t *buff, // Pointer to the data to be written
    uint8_t sector);    // sector number

```

*Program 8.2. Header file for the solid state disk device driver.*

### 8.3.4. Secure digital card interface

The **Secure Digital Memory Card** (SDC) is a popular standard for data storage in embedded systems. The SDC is an example of a high-speed I/O device, and normally we would interface the SDC using DMA synchronization. However, when interfacing to the TM4C/MSP432, we will use busy-wait synchronization with the understanding that peak bandwidth will be limited by software and not SDC performance. If we wished to improve performance, then DMA synchronization could be used. The SDC is upward-compatible to MULTI-MEDIA CARD (MMC) so that the SDC-compliant interfaces can also use an MMC with an appropriate adapter. There are also smaller versions, such as MINISD and MICROSD, where the differences are in the connector rather than the electrical specification. The card itself has a microcontroller in it. The flash memory operations, such as erasing, reading, and writing, are performed on this microcontroller. The data is transferred between the memory card and the host controller as 512-byte blocks. In this way, the SDC can be viewed like a generic hard disk drive. In other words, the low-level drivers perform block reads and writes. A 2-gibibyte SDC will have over 4 million ( $2^{31}/2^9$ ) blocks, and the low-level driver will allow you to read or write any of these blocks. Program 8.3 shows a possible header file for such a low-level software interface. The implementation of this SDC driver can be found on the book web site as **SDC\_xxx**. The file system, written as a higher level driver, will format and partition this storage in a logical manner. You can download from the internet full-functioning SDC drivers and FAT16 file system for most microcontrollers. The FAT16 file system will allow data exchange between the microcontroller and a personal computer. The FAT32 is defined for only high capacity ( $\geq 4\text{G}$ ) cards. However, this section will serve as an introduction providing the basic ideas and fundamental theories. The file systems described in the next section will be a lot simpler than FAT16.

The SDC software driver is similar to the driver for the internal flash memory presented in the last section. The **eDisk\_Init** function must be called once. **eDisk\_ReadBlock** is used to read 512 bytes of data from the SDC into RAM. **eDisk\_WriteBlock** is used to write 512 bytes of data from RAM into the

SDC. The write block function will perform the two step operating of erasing and then programming.

```
//***** eDisk_Init *****  
// Initialize the interface between microcontroller and the SD card  
// Inputs: drive number (only drive 0 is supported)  
// Outputs: status  
// STA_NOINIT 0x01 Drive not initialized  
// STA_NODISK 0x02 No medium in the drive  
// STA_PROTECT 0x04 Write protected  
// since this program initializes the disk, it must run with  
// the disk periodic task operating  
DSTATUS eDisk_Init(BYTE drive);  
  
//***** eDisk_ReadBlock *****  
// Read 1 block of 512 bytes from the SD card (write to RAM)  
// Inputs: pointer to an empty RAM buffer  
// sector number of SD card to read: 0,1,2,...  
// Outputs: result  
// RES_OK 0: Successful  
// RES_ERROR 1: R/W Error  
// RES_WRPRT 2: Write Protected  
// RES_NOTRDY 3: Not Ready  
// RES_PARERR 4: Invalid Parameter  
DRESULT eDisk_ReadBlock (  
BYTE *buff, // Pointer to the data buffer into which to store  
DWORD sector); // sector number to read from  
  
//***** eDisk_WriteBlock *****  
// Write 1 block of 512 bytes of data to the SD card (read from RAM)  
// Inputs: pointer to RAM buffer with information  
// sector number of SD card to write: 0,1,2,...  
// Outputs: result  
// RES_OK 0: Successful  
// RES_ERROR 1: R/W Error  
// RES_WRPRT 2: Write Protected  
// RES_NOTRDY 3: Not Ready  
// RES_PARERR 4: Invalid Parameter  
DRESULT eDisk_WriteBlock (  
const BYTE *buff, // Pointer to the data to be written  
DWORD sector); // sector number
```

*Program 8.3. Header file for the SDC driver (SDC\_XXX).*

With a 32-bit sector number we could support disk up to  $2^{32} \cdot 2^9$  bytes or 2 tibabytes.



Figure 8.13 shows the connector pin-out and interface. The SDC has 9 to 12 contact pads, including four pins that comprise the synchronous serial interface. **MOSI**, **MISO** and **Sclk** are the usual SPI signals, and **CS** line can be implemented with any regular output pin. The three contacts are assigned for power supply. The SDC works at supply voltages from 2.7 to 3.6 V, The current consumption can reach up to 15 mA in standby and 50 mA during operation. Some SD card connectors provide an additional pin to let the software know whether or not a SDC is inserted into the slot.

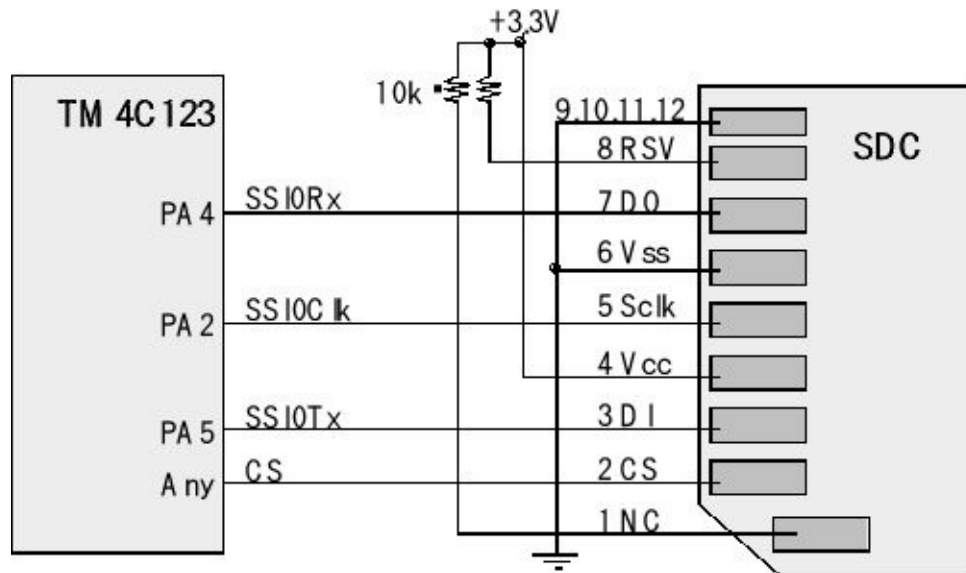


Figure 8.13. MicroSD connector(Digikey WM3288CT-ND) and TM4C interface.

There are three possible modes to interface the SD card: SD 4-bit mode, SD 1-bit mode, and SPI mode. The communication protocol for the SPI mode is simple compared to the native SD modes. Therefore, the SPI mode is suitable for low-cost embedded applications. In SPI mode, the pin 7 **DO** is always an output of the SDC, and pin 2 **DI** is always an input. Data are transferred in a byte-oriented synchronous serial fashion. The command frame from the microcontroller to the SDC is a fixed-length, six-byte packet shown in Figure 8.14. When a command frame is transmitted to the card, a response to the command (R1, R2, or R3) will eventually come from the card. The microcontroller must continue to send 0xFF frames to **DI** and receive frames from **DO**, until it receives a valid response. The command response time is 0 to 8 SPI frames (labeled as **NCR** in Figure 8.14). The **CS** signal must be held low during the entire transaction (command, response, and data transfer if exist). The 7-bit CRC field is optional in SPI mode, but it is required as a bit field to compose a command frame. The **DI** signal must be kept high during read transfer.

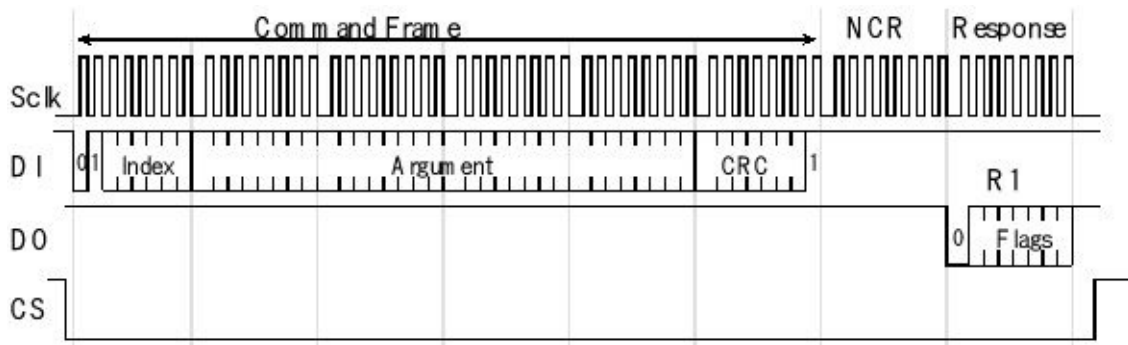


Figure 8.14. SDC command frame.

In SPI mode, data shift and data latch are done opposite clock edges respectively. There is an advantage that when shift and latch operations are separated, critical timing between two operations can be avoided. Therefore, timing consideration for IC design and board design can be relieved. The SD card uses CPOL=0, CPHA=0 mode as shown in Figure 8.15.

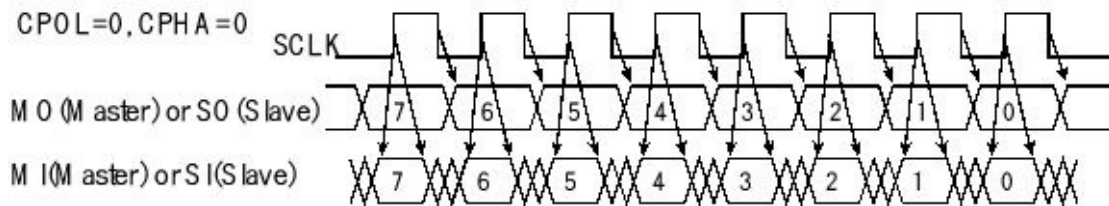


Figure 8.15. SPI CPOL = 0, CPHA = 0 mode.

There are many SD commands, some of which are shown in Table 8.1. For details on all commands, please refer to the *SDA - SD Card Association* at <http://www.sdcard.org/>. There are three command response formats: R1, R2, and R3, depending on the command index. Response R1 is 8 bits long and is returned for most commands. The R1 response has seven status bits, and a value of 0x00 means successful. Bit 6 is a parameter error, bit 5 is an address error, bit 4 is an erase sequence error, bit 3 is a communication CRC error, bit 2 is an illegal command, bit 1 is an erase reset, and bit 0 means the SDC is in the idle state. Most cards cannot change the block size, and it is fixed at 512 bytes.

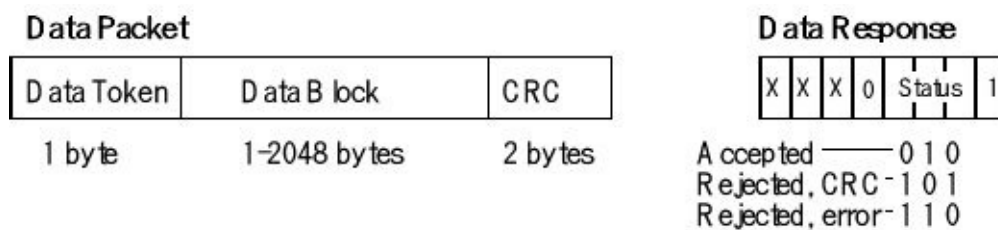
Index	Argument	Response	Data	Description
0	None	R1	No	Software reset
1 or 41	None	R1	No	Initiate initialization process
16	Block length[31:0]	R1	No	Change R/W block size
17	Address[31:0]	R1	Yes	Read a block
18	Address[31:0]	R1	Yes	Read multiple blocks
24	Address[31:0]	R1	Yes	Write a block
25	Address[31:0]	R1	Yes	Write multiple blocks

58	None	R3	No	Read OCR
----	------	----	----	----------

**Table 8.1. SD commands.**

After power on reset, the SDC enters its native operating mode. To put the SDC in SPI mode, the following procedure must be performed. After the supply voltage reaches at least 2.2 V, wait at least one more millisecond. To initialize we need to set **DI** and **CS** high and send 74 or more clock pulses to **Sclk**. After this, the card will become ready to accept native commands. We set the SPI clock rate between 100 and 400 kHz and then send an **Index=0** command with **CS** low to reset the card. The card samples the **CS** signal when an **Index=0** command is received. If the **CS** signal is low, the card enters SPI mode. Since the **Index=0** command must be sent as a native command, the CRC field must have a valid value. Once the card enters SPI mode, the CRC feature is disabled, and the CRC is not checked, so that the command transmission routine can be written with the hardcoded CRC value that is valid for only this command. When the **Index=0** is accepted, the card will enter idle state and sends an R1 response with the idle bit (0x01).

In idle state, the card accepts only commands with index values of 0, 1, 41, and 58. Any other commands will be rejected. Command **Index=58** allows you to check the working voltage range. Response R3 is an R1 plus information about the supply voltage. If the supply voltage is out of range, the card must be rejected. The card initiates initialization when a command with **Index=41** is received. To poll end of the initialization, the host controller must repeatedly send commands with **Index=41** until the idle bit goes low. When the card is initialized successfully, the idle bit in the R1 response is cleared. That is, the R1 response will change from 0x01 to 0x00. The initialization process can take hundreds of milliseconds and large cards make take longer. After the idle bit is cleared, read/write commands can be sent. Command **Index=41** is recommended instead of **Index=1** for SDC. **Index=1** initiation can be tried if **Index=41** is rejected. After initialization, the SPI clock rate can be increased to optimize the read/write performance. Most SD cards can handle SPI rates of 25 MHz. The speed will be dominated by software transferring data with the SPI port. To achieve higher bandwidth, you could use a DMA interface available on many high-performance microcontrollers.



*Figure 8.15. SD data packets.*

In a transaction with data transfer, one or more data packets will be sent/received after command response. See Figure 8.16. The data block is transferred as a data packet that consists of Token, Data Block, and CRC. The token for command indices

17, 18, and 24 is \$FE. The token for command index 25 is \$FC. A logic analyzer trace for a single-block read is shown in Figure 8.16. The resolution on the plot is not enough to see all the **Sclk** pulses. However, we see the **CS** line (labeled PA4 SS) goes low and remains low for the entire transaction. The microcontroller begins by sending an **Index=17** read block command. The argument for this command will be the sector address from which to read. The command response will be R1 with a value of 0x00, which means okay. Next, the microcontroller sends many frames (300  $\mu$ s on this system) waiting for the SDC. The last half of the transfer is a data packet being sent from the SDC to the microcontroller containing the 512 bytes read from that sector. On this system, it took 535  $\mu$ s to read one block. If any error occurred during the read operation, an error token will be returned instead of data packet.

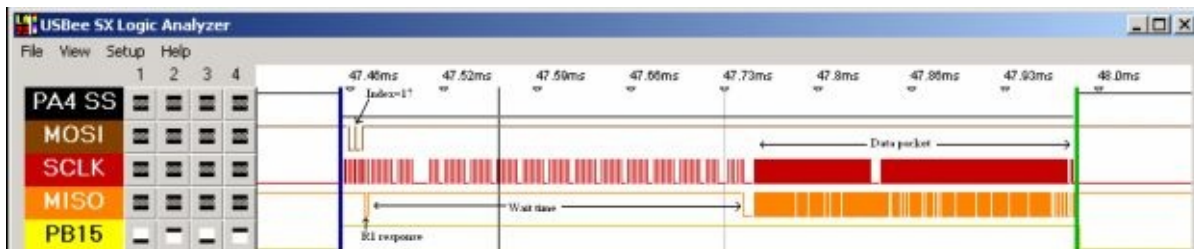


Figure 8.16. Single block read packet.

To write a block, the controller sends a write command. If the response R1 is 0x00, the microcontroller sends a data packet to the card after an eight-clock pause. The write data packet has the same format as read data packet. The CRC field can have any value unless the CRC function is enabled. When a data packet has been sent, the card responds a Data Response immediately following the data packet.

Original CD drives could read data at 150 kilobytes per second, and as faster drives arrived, manufacturers referred to their read speeds as a multiple of the original speed, referred to as X. Therefore, a 2X CD drive reads data at 300 kilobytes/sec. For DVDs the speeds are 9 times faster than CDs. I.e., a 1X DVD can read/write at 1,385,000 bytes/sec. Therefore, a 16X DVD can transfer at 16 times faster than a 1X DVD. SD Cards and SDHC Cards have Speed Class Ratings defined by the SD Association. The SD Speed Class Ratings specify the following minimum write speeds based on "the best fragmented state where no memory unit is occupied": ([www.SDCard.org](http://www.SDCard.org)). Because of the software overhead in the microcontroller, the transfer rates to the SDC will be much slower than the maximum. Table 8.2 shows example transfer rates or bandwidth for various mass-storage devices. Under most situations the size of the data block transferred is fixed. The time to locate the physical location is called the **seek time**. Although seek time has a significant impact on the disk performance, it does not affect the latency or bandwidth parameters. The bandwidth depends on the rotation speed of the disk and the information density on the medium. The transfer rates vary according to the physics of the drive.

Drive type	Bandwidth in mebibytes/sec
SATA channel	300

7200 RPM hard drive	70
16X DVD	22
52X CD-ROM	7.8
Class 2 SD card	2
Class 4 SD card	4
Class 6 SD card	6
1X CD-ROM	0.15

**Table 8.2. Bandwidth for various mass storage devices.**

Because the SDC driver functions posted on the book web site use busy-wait synchronization, actually speeds for this systems using these drivers will be much slower than the transfer rates presented in the above table.

## 8.4. Simple File System

In this section, we develop a file system that would be appropriate for implementation with an SD card used for storage. In order to implement this file system, you would need to have physical layer eDisk driver functions for the SD card. There are a couple of projects for the TM4C123 that have implementations for this physical layer. The second example includes both a low-level eDisk and a high-level FAT16 file system for the SD card.

[http://users.ece.utexas.edu/~valvano/arm/SDC\\_4C123.zip](http://users.ece.utexas.edu/~valvano/arm/SDC_4C123.zip)

[http://users.ece.utexas.edu/~valvano/arm/SDCFile\\_4C123.zip](http://users.ece.utexas.edu/~valvano/arm/SDCFile_4C123.zip)

### 8.4.1. Directory

The first component of the file system is the **directory**, as shown in Figure 8.17. In this system, the sector size is 512 bytes. In order to support disks larger than 32 Megabytes, 32-bit sector pointers will be used. The directory contains a mapping between the symbolic filename and the physical address of the data. Specific information contained in the directory might include the filename, the number of the first sector containing data, and the total number of bytes stored in the file. One possible implementation places the directory in sector 0. In this simple system, all files are listed in this one directory (there are no subdirectories). There is one fixed-size directory entry for each file. A filename is stored as an ASCII string in an 8-byte array. A null-string (first byte 0) means no file. Since the directory itself is located in sector 0, zero can be used as a null-sector pointer. In this simple scheme, the entire directory must fit into sector 0, the maximum number of files can be calculated by dividing the sector size by the number of bytes used for each directory entry. In Figure 8.17, each directory entry is 16 bytes, so there can be up to  $512/16 = 32$  files. We will need one directory entry to manage the free space on the disk, so this disk format can have up to 31 files.

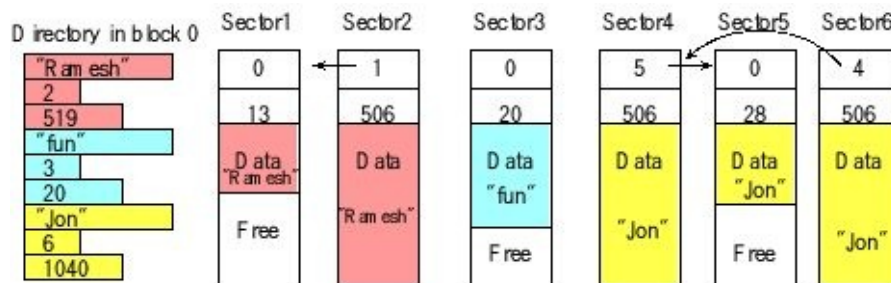


Figure 8.17. Linked file allocation with 512-byte sectors.

Other information that one often finds in a directory entry includes a pointer to the last sector of the file, access rights, date of creation, date of last modification, and

file type.

## 8.4.2. Allocation

The second component of the file system is the **logical-to-physical address translation**. Logically, the data in the file are addressed in a simple linear fashion. The logical address ranges from the first to the last. There are many algorithms one could use to keep track of where all the data for a file belongs. This simple file system uses **linked allocation** as illustrated in Figure 8.17. Recall that the directory contains the sector number of the first sector containing data for the file. The start of every sector contains a link (the sector number) of the next sector, and a byte count (the number of data bytes in this sector). If the link is zero, this is last sector of the file. If the byte count is zero, this sector is empty (contains no data). Once the sector is full, the file must request a free sector (empty and not used by another file) to store more data. Linked allocation is effective for systems that employ sequential access. Sequential read access involves two functions similar to a magnetic tape: rewind (start at beginning) and read the next data. Sequential write access simply involves appending data to the end of the file. Figure 8.17 assumes the sector size is 512 bytes and the filename has up to 7 characters. The null-terminated ASCII string is allocated 8 bytes regardless of the size of the string. The sector pointer and the size entry (e.g., file 'Ramesh' has 519 bytes) each require 4 bytes (32 bits). Since each data sector has a 4-byte link and a 2-byte counter, each sector can store up to 506 bytes of data.

## 8.4.3. Free space management

The third component of the file system is **free-space management**. Initially, all sectors except the one used for the directory are free and available for files to store data. To store data into a file, sectors must be allocated to the file. When a file is deleted, its sectors must be made available again. One simple free-space management technique uses **linked allocation**, similar to the way data is stored. Assume there are  $N$  sectors numbered from 0 to  $N-1$ . An empty file system is shown in Figure 8.18. Sector 0 contains the directory, and sectors 1 to  $N-1$  are free. You could assign the last directory entry for free-space management. This entry is hidden from the user. E.g., this free-space file cannot be opened, printed, or deleted. It doesn't use any of the byte count fields, but it does use the links to access all of the free sectors. Initially, all of the sectors (except the directory itself) are linked together, with the special directory entry pointing to the first one and the last one having a null pointer.

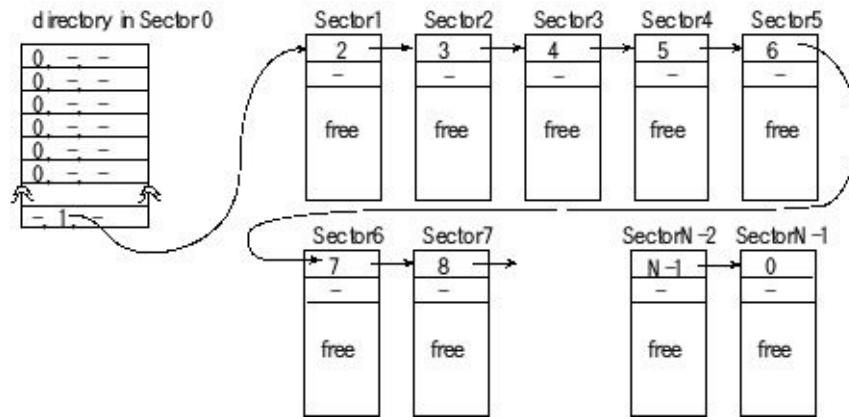


Figure 8.18. Free-space management.

When a file requests a sector, it is **unlinked** from the free space and linked to the file. When a file is deleted, all of its sectors are linked to the free space again.

**Checkpoint 8.14:** If the directory shown in Figures 8.17 and 8.18 allocated 6 bytes for the filename instead of 10, how many files could it support?



---

## 8.5. Write-once File System

### 8.5.1. Usage

Even though the previous approaches were indeed simple, we can simplify the file system even more if we make the following usage restrictions/specifications:

- The 128k flash memory is erased only once;
- The act of erasing the entire flash is equivalent to “formatting” the disk;
- The disk is partitioned into 256 sectors of 512 bytes/sector;
- We can append data to a file but cannot delete data or files;
- We append data to a file in chunks of 512 bytes;
- We will read data in a sequential fashion;
- We assign file names as single 8-bit numbers (0 to 254);
- We limit the file system to a maximum of 255 files;
- We will mount (initialize the driver) the file system on startup;
- We will call flush (backup to disk) the file system before powering down.

One sector will be reserved for the operating system to manage the directory and allocation scheme and the other 255 sectors will contain data. Depending on the debugger settings, loading the program into flash may erase the entire flash. Alternately, you could explicitly erase the flash in the debugger, or you could call the **OS\_File\_Format** function. These erase events will serve to “format” the disk. All 255 data sectors will be free and the file system will have no files. However, hitting the reset button or powering up the system should not erase the disk.

While using this disk we could have 255 individual files, each with one sector. We could have 51 files each with 5 sectors. Alternately, we could have one file with 255 sectors. Any combination is possible where the number of files is less than or equal to 255, and the total allocated sectors is also less than or equal to 255.

There will be a function, **OS\_File\_New**, which will return the file number of an empty file. This function will fail if there are no more files left, because there are already 254 files created, or if there are no free sectors, because the disk is full.

```
/**OS_File_New***/  
// Returns a file number of a new file for writing  
// Inputs: none
```

```

// Outputs: number of a new file
// Errors: return 255 on failure or disk full
uint8_t OS_File_New(void);

```

To check the status of a file, we can call **OS\_File\_Size** . This function returns the number of sectors allocated to this file. If the size is zero, this is an empty file.

```

//*****OS_File_Size*****
// Check the size of this file
// Inputs: num, 8-bit file number, 0 to 254
// Outputs: 0 if empty, otherwise the number of sectors
// Errors: none
uint8_t OS_File_Size(uint8_t num);

```

To write data to an existing file we need to specify the file number into which we will store the data. The write data function will allocate another sector to the file and append 512 bytes of new data to the file. The input parameters to **OS\_File\_Append** are the file number and a sector of 512 bytes of data to write. This function will fail if there are no free sectors (disk full).

```

//*****OS_File_Append*****
// Save 512 bytes into the file
// Inputs: num, 8-bit file number, 0 to 254
// buf, pointer to 512 bytes of data
// Outputs: 0 if successful
// Errors: 255 on failure or disk full
uint8_t OS_File_Append(uint8_t num, uint8_t buf[512]);

```

To read data from a file we call **OS\_File\_Read** . The three parameters to this function are the file number, the location, and a pointer to RAM. The **location** parameter defines the logical address of the data in a file. Location 0 will access the first sector of the file. For example, if a file has 5 sectors, the **location** parameter could be 0, 1, 2, 3, or 4. The read data function will copy 512 bytes of data from the file into the RAM buffer. This function will fail if this file does not have data at this location.

```

//*****OS_File_Read*****
// Read 512 bytes from the file
// Inputs: num, 8-bit file number, 0 to 254
// location, logical address, 0 to 254
// buf, pointer to 512 empty spaces in RAM
// Outputs: 0 if successful
// Errors: 255 on failure because no data
uint8_t OS_File_Read(uint8_t num, uint8_t location,
uint8_t buf[512]);

```

We will load into RAM versions of the directory and the FAT when the system starts. When we call **OS\_File\_Flush** the RAM versions will be stored onto the disk. Notice that due to the nature of how this file system is designed, bits in the directory and FAT never switch from 0 to 1. We can either call this function periodically or call it once just before power is removed from the system.

```

//*****OS_File_Flush*****
// Update working buffers onto the disk
// Power can be removed after calling flush
// Inputs: none
// Outputs: 0 if success
// Errors: 255 on disk write failure
uint8_t OS_File_Flush(void);

```

Depending on the debugger settings, downloading software may erase the flash. When the flash is erased, the disk in essence is formatted, because we defined the all ones state as empty. However, if one wishes to erase the entire disk removing all data and all files, one could call **OS\_File\_Format**. This function will erase the flash from 0x00020000 to 0x0003FFFF. Program 8.4 shows the implementation for the TM4C123. It simply erases all blocks from 0x00020000 to 0x0003FFFF. Notice that this implementation skips the eDisk layer and directly calls the physical layer.

```

//*****OS_File_Format*****
// Erase all files and all data
// Inputs: none
// Outputs: 0 if success
// Errors: 255 on disk write failure
uint8_t OS_File_Format(void){
    uint32_t address;
    address = 0x00020000; // start of disk
    while(address <= 0x00040000){
        Flash_Erase(address); // erase 1k block
        address = address+1024;
    }
}

```

*Program 8.4. TM4C123 version of formatting.*

**Checkpoint 8.15:** The physical block size on the MSP432 is 4096 bytes. How would you modify **OS\_File\_Format** for the MSP432?

## 8.5.2. Allocation

There are many possible solutions, but we chose FAT allocation because it supports appending to an existing file. FAT supports many small files or one large file. Because there are 256 sectors we will use 8-bit sector addresses. Because we will

define a completely erased flash as “formatted”, we will use the sector address 255=0xFF to mean null-pointer, and use sector number 255 as the directory. To implement a FAT with this disk, we would need only 255 bytes. Since the sector is 512 bytes we can use 256 bytes for the directory and the other 256 bytes for the FAT. Notice that sectors are allocated to files, but never released. This means we can update the FAT multiple times because bits are all initially one (erased) and programmed to 0 once, and never need to be erased again.

Since the files are identified by number and not name, the directory need not store the name. Rather, the directory is a simple list of 255 8-bit numbers, containing the sector number of its first sector. Notice there is exactly one directory entry for each possible file. If this sector number is 255, this file is empty. Similarly, the FAT is another simple list of 255 8-bit numbers. However, a 255 in the FAT may mean a free sector or the last sector of a file. Notice there is one entry in the FAT for each data sector on the disk. Figure 8.19 shows the disk after formatting. Each rectangle in the disk figure represents a 512-byte data sector. The directory and FAT are both stored in sector number 255.

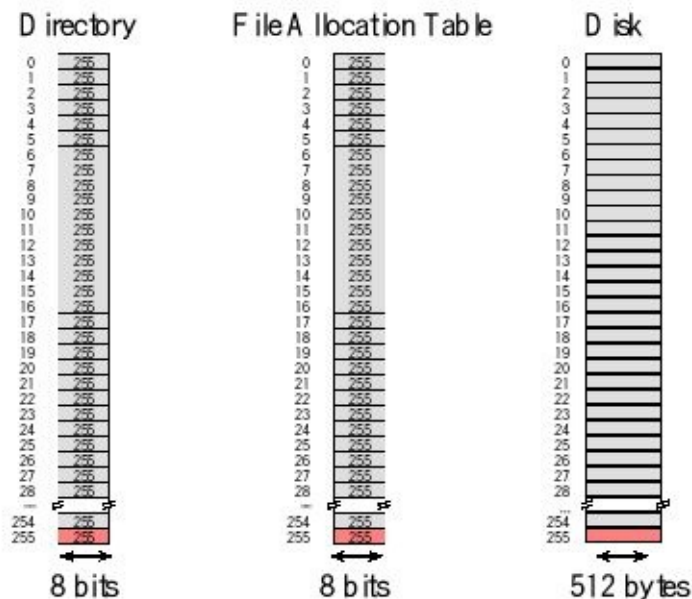


Figure 8.19. Empty disk on the write-once file system.

If we ask for a new file, the system will return a number from 0 to 254 of a file that has not been written. In other words, **OS\_File\_New** will return the number of an empty file. If we execute the following when the disk is empty, **OS\_File\_New** will return a 0 (n=0), and the eight calls to **OS\_File\_Append** will store eight sectors on the disk, see Figure 8.20.

```
n = OS_File_New();
OS_File_Append(n,buf0);
OS_File_Append(n,buf1);
OS_File_Append(n,buf2);
```

```

OS_File_Append(n,buf3);
OS_File_Append(n,buf4);
OS_File_Append(n,buf5);
OS_File_Append(n,buf6);
OS_File_Append(n,buf7);

```

In this example, the variables **n,m,p** are simple global variables containing the file numbers we are using. The parameters **buf0-buf9**, **dat0-dat4**, **arr0-2** represent RAM buffers with 512 bytes of data. Having 18 buffers we do not imply we needed a separate RAM buffer for every sector on the disk, but rather to differentiate where data is stored on the disk. In other words, the use of 18 different RAM buffers was meant to associate the 18 calls to OS\_File\_Append with the corresponding 18 sectors used on the disk. Because of the limited RAM on the microcontroller, normally we will limit the number of RAM buffers.

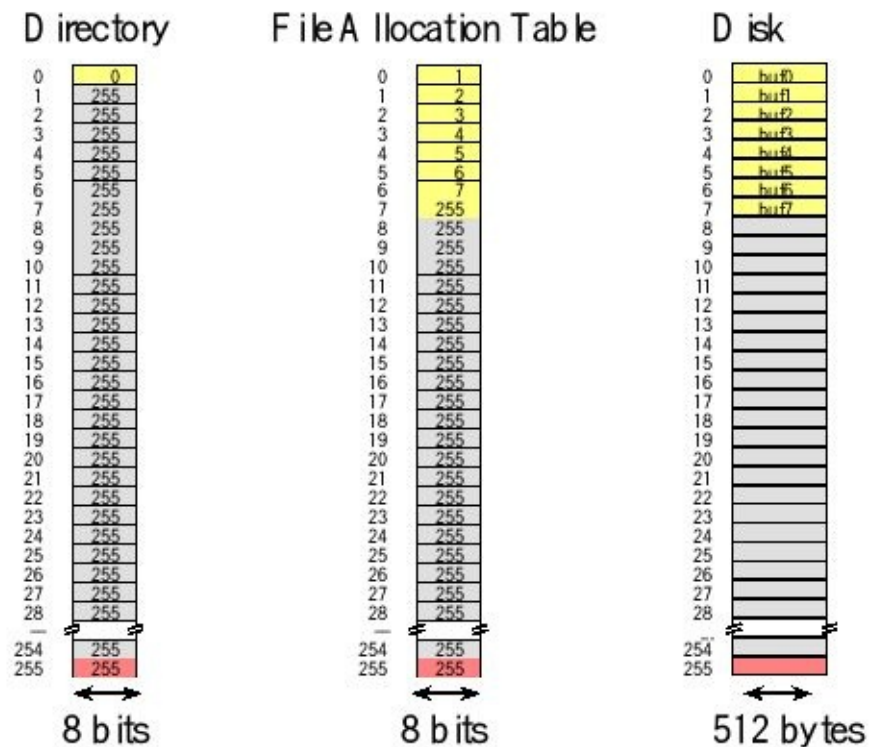


Figure 8.20. A disk with one file, this file has 8 sectors.

If we were to continue this example and execute the following, there would now be 3 files on the disk occupying 18 sectors. See Figure 8.21.

```

m = OS_File_New();
OS_File_Append(m,dat0);
OS_File_Append(m,dat1);
OS_File_Append(m,dat2);
OS_File_Append(m,dat3);
p = OS_File_New();

```

```

OS_File_Append(p,arr0);
OS_File_Append(p,arr1);
OS_File_Append(n,buf8);
OS_File_Append(n,buf9);
OS_File_Append(p,arr2);
OS_File_Append(m,dat4);

```

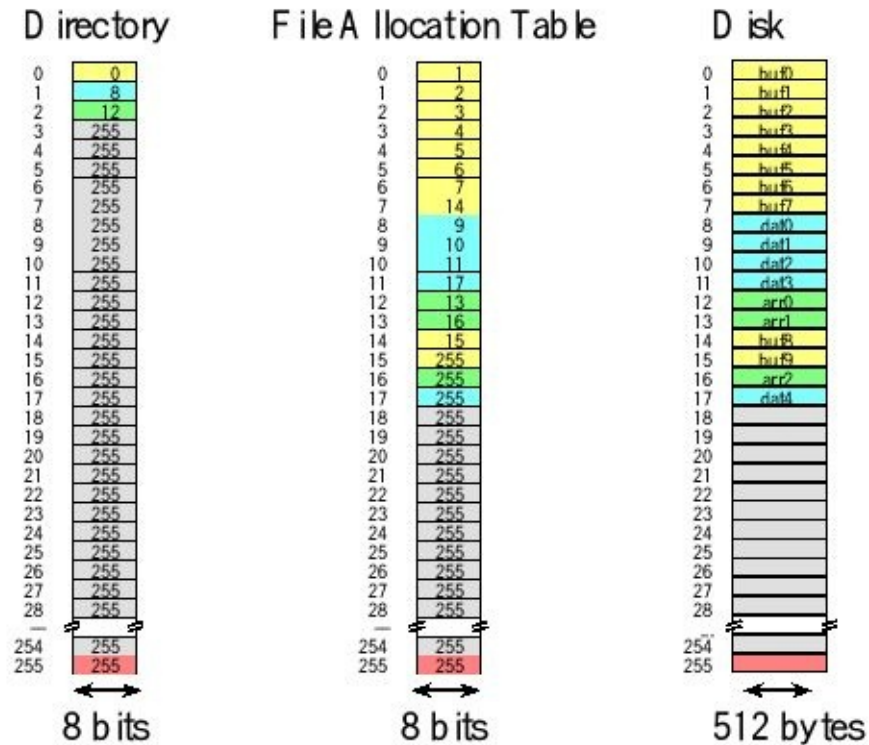


Figure 8.21. A disk with three files, file 0 has 10 sectors, file 1 has 5 sectors and file 2 has 3 sectors.

Notice that we limit usage to adding data to the disk is chunks of 512 bytes. As mentioned earlier we will never delete a file, nor will we delete parts of a file previously written. Furthermore, we always append to the end of a file, which means we never move data of a file from one place on the disk to another.

### 8.5.3. Directory

We will read the directory/FAT into RAM on startup. We need to be able to write the directory to the disk multiple times. We will write the directory/FAT each time we close a file and before removing power. Figure 8.22 shows one possible implementation of the process to create a new file. This function will return the file number (0 to 254) of a file not yet written to.

Since files are never deleted, this function will return file numbers in a 0, 1, 2, ... order. Once there are 255 files on the disk, no more files can be created.

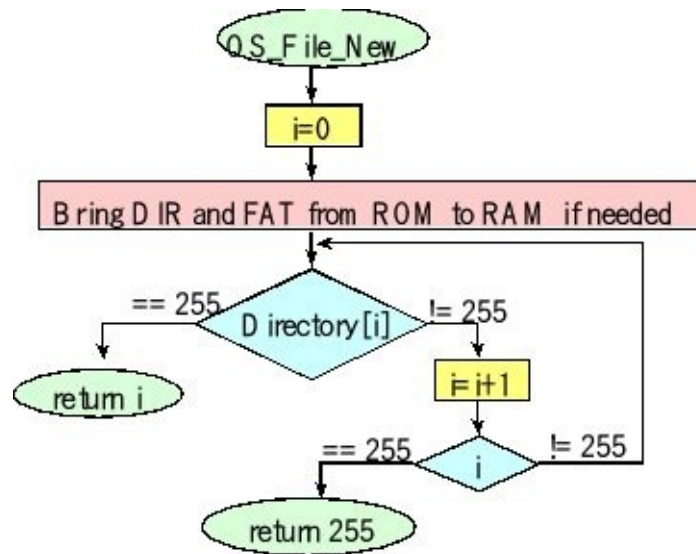


Figure 8.22. Software flowchart for OS\_File\_New. Returning with a 255 means fail because the disk already has 254 files. The only way for this function to fail is if the disk has 254 files, and each file is one sector.

This simple file system assumes you append some data after you create a new file and before you create a second new file. The following shows a proper use case of creating multiple files:

```

n = OS_File_New();    // create a new file
OS_File_Append(n,stuff); // add to n
m = OS_File_New();    // second file
OS_File_Append(m,other); // add to m
  
```

If you violate this assumption and execute the following code, then files n and m will be one file. I.e., n will equal m.

```

n = OS_File_New();    // create a new file
m = OS_File_New();    // second file
OS_File_Append(n,stuff); // add to n
OS_File_Append(m,other); // add to m
  
```

### 8.5.4. Append

Figure 8.23 shows one possible implementation of the function that appends a data buffer to an existing file.

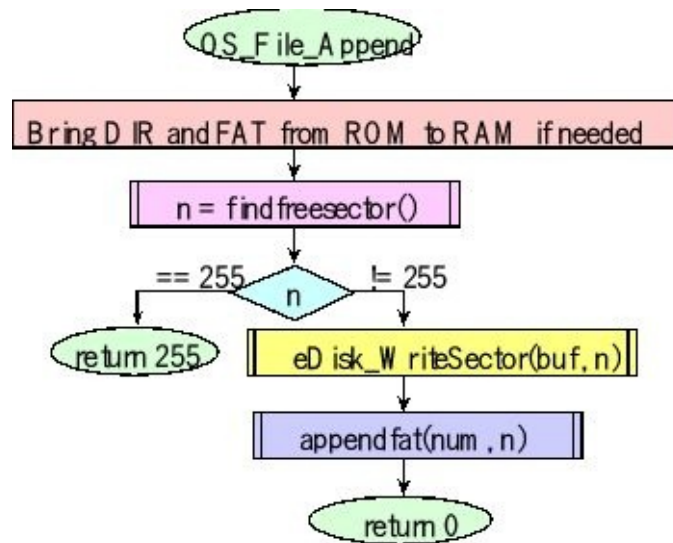


Figure 8.23. Software flowchart for OS\_File\_Append. Returning with a 255 means fail because there are no free sectors on the disk.

Figure 8.24 shows the helper function that appends the sector number (n) to the FAT link associated with file (num).

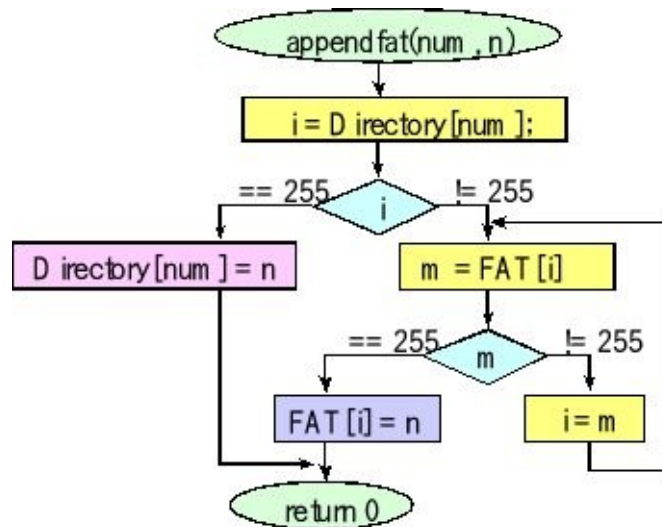


Figure 8.24. Software flowchart for the helper function appendfat.

### 8.5.5. Free space management

An entry in the FAT of 255 means that sector is free or that is the last sector of a file. However, since files are never deleted or reduced in size, there will be no external fragmentation and all free sectors exist in one contiguous chunk. In particular, if we search the FAT for the last sector of each file, find the maximum of these numbers, the first free sector is this maximum+1. The last free sector is 254. Figure 8.25 shows the helper function that finds a free sector on the disk.



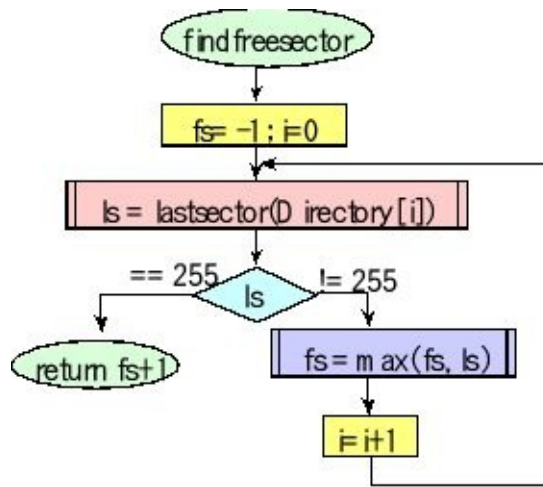


Figure 8.25. Software flowchart for the helper function findfreesector.

Figure 8.26 shows the helper function that finds the last sector of file that starts at sector.

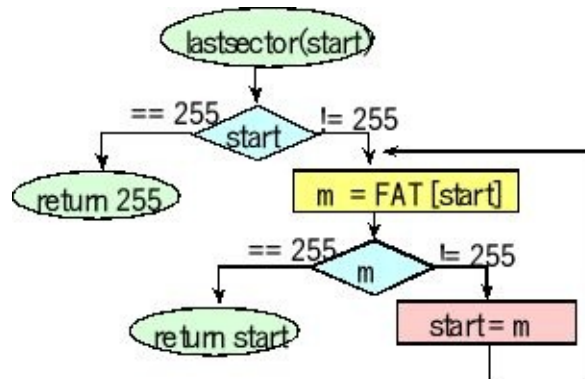


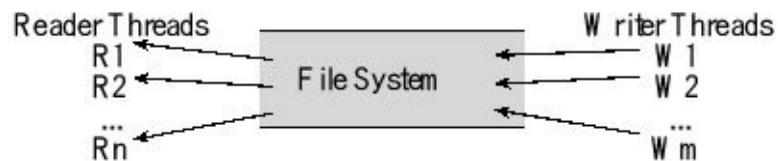
Figure 8.26. Software flowchart for the helper function lastsector.

## 8.6. Readers-Writers Problem

When threads within an OS share a common file system, synchronization will be required to prevent corrupted or inconsistent data, see Figure 8.27. Multiple readers are allowed concurrent access to the file system because readers do not modify the data. Table 8.3 shows a reader will open a file for reading, access the data, and then close the file. On the other hand, only one writer is allowed access to the data at a time. A writer thread will open the file with read/write permission, read and write to the file, and then close the file.

Reader Threads	Writer Threads
1) Execute ROpen(file)	1) Execute WOpen(file)
2) Read information from file	2) Read/write information from/to file
3) Execute RClose(file)	3) Execute WClose(file)

**Table 8.3.** Sequence of action employed by readers and writers.



*Figure 8.27. A file system can have multiple readers and multiple writers.*

The solution to the readers-writers problem uses three objects: a numerical counter called **ReadCount**, a binary semaphore called **mutex**, and another binary semaphore called **wrt**. The **ReadCount** defines how many reader threads are accessing the file system and this counter is initialized to 0. The **mutex** semaphore is used to create mutually exclusive access to shared information in **ReadCount**, and **mutex** is initialized to 1. The **wrt** semaphore allows just one writer to access the file system and **wrt** is initialized to 1. Program 8.5 shows the synchronization required to open and close files. If a reader thread is first, it will prevent writers from access by executing a wait on **wrt**. Once all readers are finished, the **wrt** semaphore is signaled. If a writer thread is first, it will prevent all other threads from accessing the file system.

ROpen wait(&mutex); ReadCount++;	RClose wait(&mutex); ReadCount--;	WOpen wait(&wrt);	WClose signal(&wrt);
if(ReadCount==1) { wait(&wrt); }	if(ReadCount==0) { signal(&wrt); }		

```
signal(&mutex);
```

```
signal(&mutex);
```

*Program 8.5. Semaphore synchronization used to solve the readers-writers problem.*

---

## 8.7. Exercises

**8.1** For each term give a definition in 32 words or less.

- a) Free-space management   b) Linked allocation   c) Indexed allocation  
d) FAT                      e) Internal fragmentation   f) External fragmentation

**8.2** Consider a file system that uses **contiguous allocation** to define the set of blocks allocated to each file, as shown in Figure 8.28. There are 8192 bytes on this disk made up of 256 blocks, where each block is 32 bytes. This file system is used to record important “black box” information. Therefore, the file system is initialized to empty when the device is manufactured. Each time the system is turned on, a new file is created. While running important data are stored into that file (create new file, append data at the end, close file). Files are never deleted. Once a file is closed, it can be opened for reading, but it cannot be opened again for writing. Block 0 contains the directory and not available for data. Each directory entry has three fields: name, block number of the first block, and total number of bytes stored. The example in Figure 8.28 shows file A with 3 allocated blocks (1,2,3 containing 32,32,8 bytes), file B with 2 blocks (4,5 containing 32,32 bytes), and file C with 7 blocks (6,7,8,9,10,11,12 containing 32,32,32,32,32,32,8). All 32 bytes of each data block can contain data for the file.

- a) Does this file system have any external fragmentation? Justify your answer.  
b) Assume a file has  $n$  data blocks. It takes one *block read* to fetch the **directory**. On **average**, how many more *block reads* does it take to read a single byte at a random position in the file? What is the **maximum** number of additional *block reads* that it takes to read a single byte in the file (worst case)?  
c) Describe a simple mechanism to manage free blocks in this system. Be as explicit as possible, describing how many bytes in the directory are needed to manage the free space. Describe what the free space looks like after the disk is erased/formatted. Describe what the free space looks like when the disk is full.  
d) File names are a single character. How many files can be stored? Justify your answer.  
e) Assume you have  $n$  files each with of random size. Quantify the number of wasted bytes due to internal fragmentation. You may assume  $n$  is less than the number determined in part d).

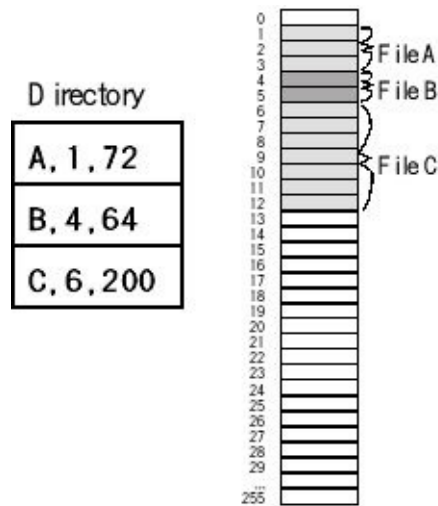


Figure 8.28. File system for Exercise 8.2.

8.3 Consider a file system that uses contiguous allocation, as illustrated by Figure 8.29. The block size is 32 bytes and all 256 blocks can be used to store data. The directory is not stored on the disk. Each directory entry contains the file name (e.g., A, B, C), the start block (e.g., File B starts at block 4), and the number of blocks used in the file (e.g., File C has 5 blocks). The file sizes are always a multiple of 32 bytes. I.e., a file can contain only 32, 64, 96, ..., 8192 bytes. For example, File A is  $3 \times 32 = 96$  bytes, File B is  $2 \times 32 = 64$  bytes and File C is  $5 \times 32 = 160$  bytes. Does this system have internal fragmentation? Explain your answer.

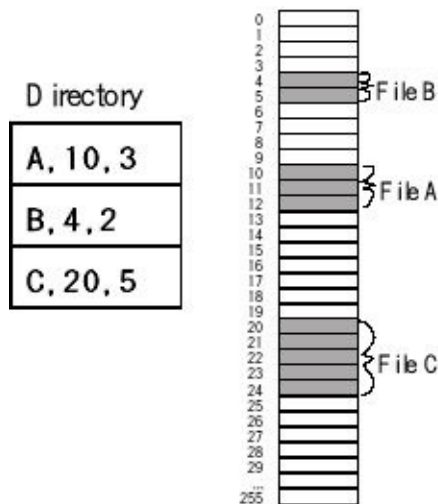


Figure 8.29. File system for Exercise 8.3.

8.4 Consider a file system that manages a 16 Megabyte ( $2^{24}$  bytes) EEPROM storage for a battery-powered embedded system. You are free to select from a range of EEPROM chips with different block sizes. The block size can be any power of 2 from 1 to  $2^{24}$  bytes. **Chip<sub>n</sub>** has a total of 16 Megabytes with block size  $2^n$  bytes. **Chip<sub>n</sub>** can perform a  $2^n$  byte block-write operation in 1 ms regardless of block size. For bandwidth reasons, therefore, you wish to choose a large block size. A block will be completely allocated to a file (you are not allowed to split one block between two

files). 16 bytes of each block are used by the file system to manage pointers, type, size, and free space. However, if the file were to contain 1 byte of data, an entire block would be allocated, and the remaining  $2^n - 17$  bytes would be wasted. File sizes in this system are uniformly distributed from 50,000 to 150,000 bytes (this means any file size from 50,000 to 150,000 bytes is equally likely with an average size of 100,000 bytes). **You are asked to choose the largest block size with the constraint that the average internal fragmentation be less 5% of the total number of bytes stored.** Show your work.

**D8.5** One way to manage free-space on a disk is to implement a **bit vector**. Each block is 32 bytes long, and there are 256 blocks. For each block on our 8-kibibyte disk, there will be a single bit specifying whether the block is free (1) or allocated. In C, we can define 256 bits as a byte-array with 32 entries.

```
uint8_t BitVector[32]; // 256 bits
```

Similar to the directory, the BitVector will exist both in RAM, as the above C definition, and on the disk as block 1. The format operation will initialize 254 of these bits to 1, performing:

```
BitVector[0] = 0x3F; // blocks 0,1 used (directory, BitVector)  
for(i=1;i<32;i++) BitVector[i]=0xFF; // blocks 8-255 are free  
eDisk_WriteBlock(BitVector,1); // update disk copy
```

a) Write a helper function that allocates a free-block updating the disk copy of BitVector.

```
// allocate a free block, returns a block number of a free block  
// Output: block number 2 to 255 if successful and 0 if full
```

```
uint8_t AllocateBlock(void){
```

```
    eDisk_ReadBlock(BitVector,1); // fresh RAM copy
```

b) Write a helper function that deallocates a block updating the disk copy of BitVector.

```
// deallocate a free block
```

```
// Input: block number 2 to 255
```

```
void DeallocateBlock(uint8_t blockNum){
```

```
    eDisk_ReadBlock(BitVector,1); // fresh RAM copy
```

**8.6** Consider a file system that uses a **file translation table (FTT)** to define the set of blocks allocated to each file. There are 65536 bytes on this disk made up of 256 blocks, where each block is 256 bytes. Block 0 contains the directory and is not available for data. Each file has its own **FTT**, which is a null-terminated list of block numbers assigned to the file. Figure 8.30 shows a file with 4 allocated blocks, with the first block at 12, and the last block at 22. The directory entry includes the file

name, the total number of bytes, and the block number of its **FTT**. All 256 bytes of each data block can contain data for the file. For example, the figure shows a file with 1024 bytes of data, stored in 5 blocks (**FTT** and 4 data blocks).

- Does this file system have any external fragmentation? Justify your answer.
- Assume a file has  $n$  data blocks. It takes one *block read* to fetch the **FTT**. On **average**, how many more *block reads* does it take to read a single byte at a random position in the file? What is the **maximum** number of additional *block reads* that it takes to read a single byte in the file (worst case)?
- Consider the linked allocation scheme. Assume the directory is in memory and the file has  $n$  data blocks. On **average**, how many *block reads* does it take to read a single byte at a random position in the file? What is the **maximum** number of *block reads* that it takes to read a single byte in the file (worst case)?
- Assume you are given the following function that reads a 256-byte block from disk

```
int eDisk_ReadBlock(uint8_t *pt, // result returned by reference
uint8_t blockNum);           // which block to read
```

Write a C function that returns a byte from a file at a random location. Do not worry about error handling (e.g., **eDisk\_ReadBlock** error or address too big). The inputs to the function are **numFTT** (the block number of the file's **FTT**) and **address** (the byte address, where 0 is the first byte, 1 means second byte etc.). You can use two buffers.

```
uint8_t FTTbuf[256]; // place to store FTT
uint8_t Databuf[256]; // place to store data
```

The prototype of the C function you have to write is

```
uint8_t eFile_Read(uint8_t numFTT, uint16_t address);
```

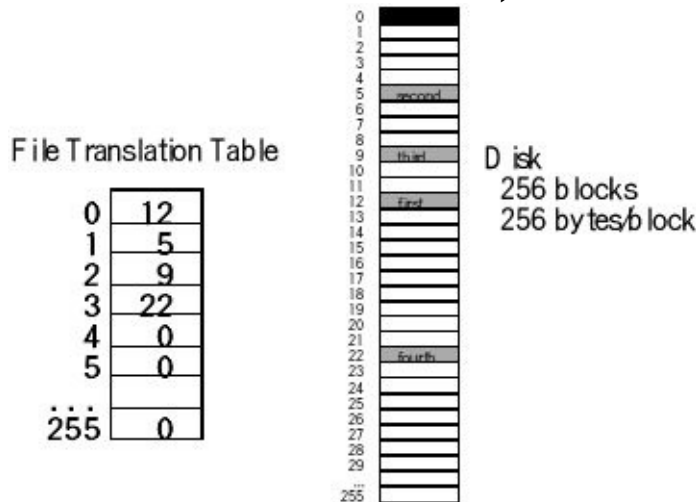


Figure 8.30. File system for Exercise 8.6.

# 9. Communication Systems

## Chapter 9 objectives are to:

- Introduce basic concepts of networks
- Describe the controller area network (CAN) protocol
- Present fundamentals and implementation of Bluetooth Low Energy (BLE)
- Introduce Ethernet, Wireless, and the Internet of Things

The goal of this chapter is to provide a brief introduction to communication systems. Communication theory is a richly developed discipline, and much of the communication theory is beyond the scope of this book. Nevertheless, the trend in embedded systems is to employ multiple intelligent devices, therefore the interconnection will be a strategic factor in the performance of the system. These devices will be developed by different manufacturers, thus the interconnection network must be flexible, robust, and reliable. Consequently, this chapter focuses on implementing communication systems appropriate for embedded systems. The components of an embedded system typically combined to solve a common objective, thus the nodes on the communication network will cooperate towards that shared goal. In particular, requirements of an embedded system, in general, involve relatively low bandwidth, static configuration, and a low probability of corrupted data. In Volume 2, networks designed with serial ports and ZigBee were presented. In this chapter we will discuss CAN, Bluetooth, and Ethernet.



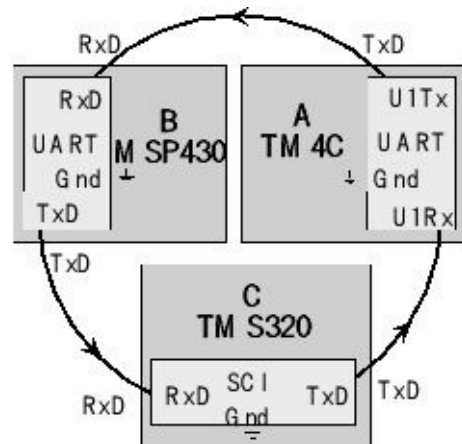
---

## 9.1. Fundamentals

### 9.1.1. The network

A **network** is a collection of interfaces that share a physical medium and a data protocol. A network allows software tasks in one computer to communicate and synchronize with software tasks running on another computer. For an embedded system, the network provides a means for distributed computing. The **topology** of a network defines how the components are interconnected. Example topologies include rings, busses and multi-hop. Figure 9.1 shows a **ring** network of three microcontrollers. The advantage of this ring network is low cost and can be implemented on any microcontroller with a serial port. Notice that the microcontrollers need not be the same type or speed. The CAN network, presented in Section 9.2, is an example of a **multi-drop bus**.

The ZigBee wireless network described in Volume 2 is a multi-hop network (duplicated in Figure 9.2). Notice that there can be multiple paths with which to route packets.



*Figure 9.1. A simple ring network with three nodes, linked using the serial ports.*

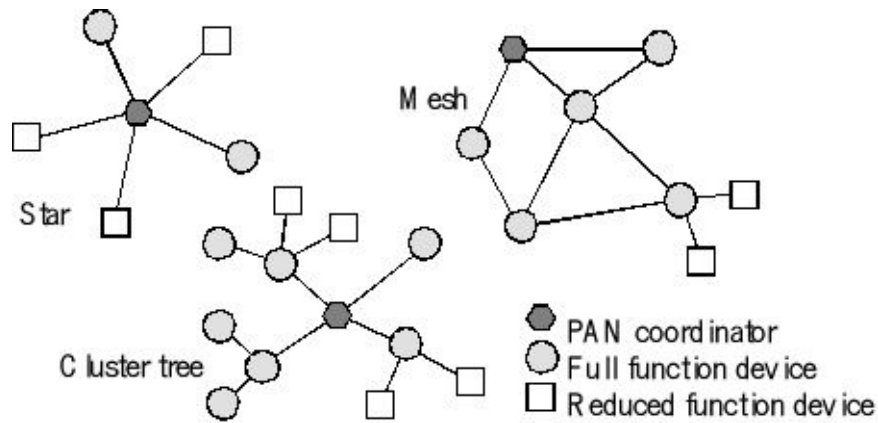


Figure 9.2. ZigBee wireless networks communicate by hopping between nodes.

In Chapter 11 of Volume 2, we considered networks with one or two layers. In this chapter, we will build on those ideas and introduce the concepts of networks with more layers and higher bandwidths.

A **communication network** includes both the physical channel (hardware) and the logical procedures (software) that allow users or software processes to communicate with each other. The network provides the transfer of information as well as the mechanisms for process synchronization. It is convenient to visualize the network in a hierarchical fashion as shown in Figure 9.3.

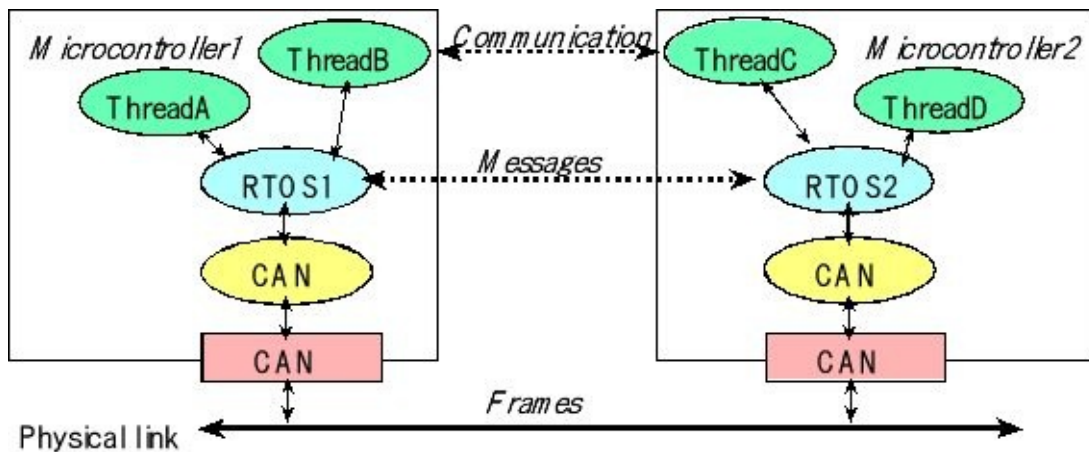


Figure 9.3. A layered approach to communication systems.

Most networks provide an **abstraction** that hides low-level details from high-level operations. This abstraction is often described as layers. The International Standards Organization (ISO) defines a 7-layer model called the **Open Systems Interconnection (OSI)**, as shown in Figure 9.4. It provides a standard way to classify network components and operations.

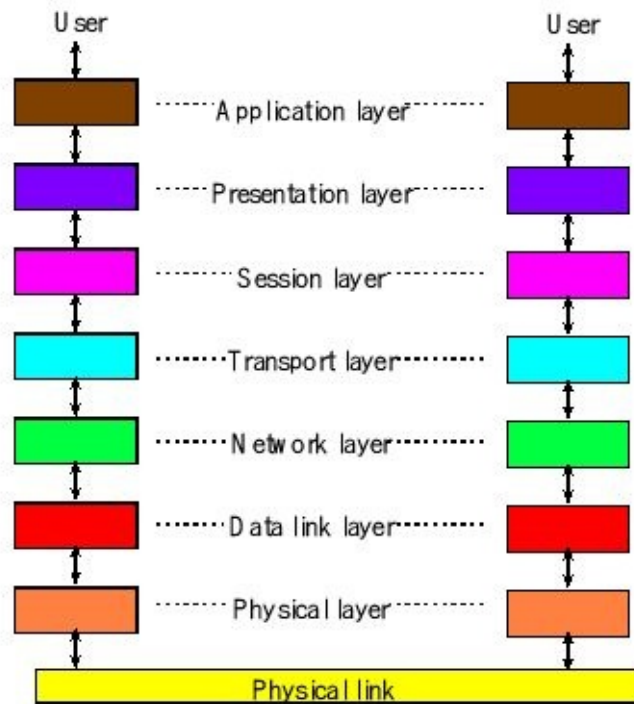


Figure 9.4. The Open Systems Interconnection model has seven layers.

The **Physical** layer includes connectors, bit formats, and a means to transfer energy. Examples include RS232, controller area network (CAN), modem V.35, T1, 10BASE-T, 100BASE-TX, DSL, and 802.11a/b/g/n PHY. The **Data link** layer includes error detection and control across a single link (single hop). Examples include 802.3 (Ethernet), 802.11a/b/g/n MAC/LLC, PPP, and Token Ring. The **Network** layer defines end-to-end multi-hop data communication. The **Transport** layer provides connections and may optimize network resources. The **Session** layer provides services for end-user applications such as data grouping and check points. The **Presentation** layer includes data formats, transformation services. The **Application** layer provides an interface between network and end-user programs.

**Observation:** Communication systems often specify bandwidth in total bits/sec, but the important parameter is the data transfer rate.

**Observation:** Often the bandwidth is limited by the software and not the hardware channel.

Many embedded systems require the communication of command or data information to other modules at either a near or a remote location. A **full duplex** channel allows data to transfer in both directions at the same time. Ethernet, SPI, and UART implement full duplex communication. In a **half duplex** system, data can transfer in both directions but only in one direction at a time. Half duplex is popular because it is less expensive and allows the addition of more devices on the channel without change to the existing nodes. CAN, I<sup>2</sup>C, and most wireless protocols implement half-duplex communication. A **simplex** channel allows data to flow in only one direction.

**Checkpoint 9.1:** In which manner to most people communicate: simplex, half duplex or full duplex?

## 9.1.2. Physical Channel

Information, such as text, sound, pictures and movies, can be encoded in digital form and transmitted across a channel, as shown in Figure 9.5. **Channel capacity** is defined as the maximum information per second it can transmit. In order to improve the effective bandwidth many communication systems will compress the information at the source, transmit the compressed version, and then decompress the data at the destination. Compression essentially removes redundant information in such a way that the decompressed data is identical (**lossless**) or slightly altered but similar enough (**lossy**). For example, a 400 pixels/inch photo compressed using the JPEG algorithm will be 5 to 30 times smaller than the original. A **guided medium** focuses the transmission energy into a well-defined path, such as current flowing along copper wire of a twisted pair cable, or light traveling along a fiber optic cable. Conversely, an **unguided medium** has no focus, and the energy field diffuses as it propagates, such as sound or EM fields in air or water. In general, for communication to occur, the transmitter must encode information as **energy**, the channel must allow the energy to move from transmitter to receiver, and the receiver must decode the energy back into the information, see Figure 9.5. In an analog communication system, energy can vary continuously in amplitude and time. A digital communication signal exists at a finite number of energy levels for discrete amounts of time. Along the way, the energy may be lost due to **attenuation**. For example, a simple  $V=I \cdot R$  voltage drop is in actuality a loss of energy as electrical energy converted to thermal energy. A second example of attenuation is an RF cable splitter. For each splitter, there will be 50% attenuation, where half the energy goes left and the other half goes right through the splitter. Unguided media will have attenuation as the energy propagates in multiple directions. Attenuation causes the received energy to be lower in amplitude than the transmitted energy.

A second problem is **distortion**. The transfer gain and phase in the channel may be function of frequency, time, or amplitude. Distortion causes the received energy to be different shape than the transmitted energy.

A third problem is **noise**. The noise energy is combined with the information energy to create a new signal. White noise is an inherent or internally generated noise caused by thermal fluctuations. EM field noise is externally generated and is coupled or added into the system. **Crosstalk** is a problem where energy in one wire causes noise in an adjacent wire. We quantify noise with **signal-to-noise ratio** (SNR), which is the ratio of the information signal power to noise power.

$$\text{SNR(dB)} = 10 \cdot \log_{10} \left( \frac{\text{Average signal power}}{\text{Average noise power}} \right)$$

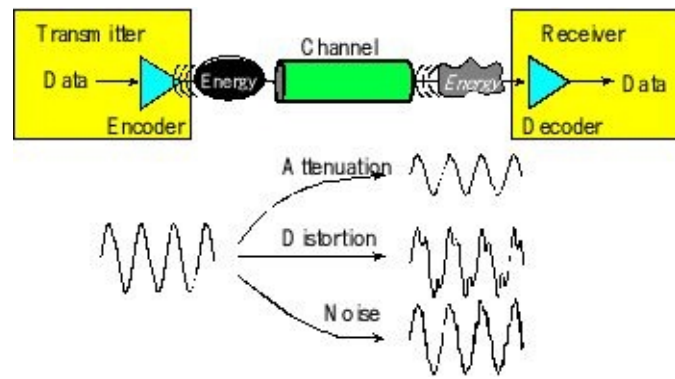


Figure 9.5. Information is encoded as energy, and errors can occur during transmission.

**Checkpoint 9.2:** Why do we measure SNR as power and not voltage?

**Checkpoint 9.3:** Why do we always have a ratio of two signals whenever we use the  $\log_{10}$  to calculate the amplitude of a signal?

**Observation:** Whenever we use the  $\log_{10}$  to calculate the amplitude of a signal, we multiply by 10 if we have a ratio of two power signals or energy signals, and we multiply by 20 if we have a ratio of two voltage signals or current signals.

We can make an interesting analogy between time and space. A communication system allows us transfer information from position A to position B. A digital storage system allows us transfer information from time A to time B. Many of the concepts (encoding/decoding information as energy, signal to noise ratio, error detection/correction, security, and compression) apply in an analogous manner to both types of systems.

**Checkpoint 9.4:** We measure the performance of a communication system as bandwidth in bits/sec. What is the analogous performance measure of a digital storage system?

Errors can occur when communicating through a channel with attenuation, distortion and added noise. If the receiver detects an error, it can send a negative acknowledgement so the transmitter will retransmit the data. The CAN, ZigBee, and Bluetooth protocols handle this detection-retransmission process automatically. Networks based on the UARTs could define and implement error detection. For example, we can add an additional bit to the serial frame for the purpose of detecting errors. With **even parity**, the sum of the data bits plus the parity bit will be an even number. The framing error in the UART can also be used to signify the data may be corrupted. The CAN network sends a **longitudinal redundancy check**, LRC, which is the exclusive or of the bytes in the frame. The ZigBee network adds a **checksum**, which is the sum of all the data. The Network Processor Interface (NPI) later in this chapter uses LRC.

There are many ways to improve transmission in the channel, reducing the probability of errors. The first design choice is the selection of the interface driver.

For example, RS422 is less likely to exhibit errors than RS232. Of course having a driver will be more reliable than not having a driver. The second consideration is the cable. Proper shielding can improve SNR. For example, Cat6 Ethernet cables have a separator between the four pairs of twisted wire, which reduce the crosstalk between lines as compared to Cat5e cable. If we can separate or eliminate the source of added noise, the SNR will improve. Reducing the distance and reducing the bandwidth often will reduce the probability of error. If we must transmit long distances, we can use a repeater, which accepts the input and retransmits the data again.

Error correcting codes are beyond the scope of this book. However, we can present two simple error correcting codes. The first error correcting code involves sending three copies of each data. The receiver will compare the three versions received and majority vote will decide which value to use. A second error correcting code uses both parity and LRC. For example, assume we wished to send the message “Ciao”. Encoded as ASCII characters the data are \$43, \$69, \$61, and \$6F. The first step is to display the binary data in 2-D.

	Byte 0	Byte 1	Byte 2	Byte 3
Bit 7	0	0	0	0
Bit 6	1	1	1	1
Bit 5	0	1	1	1
Bit 4	0	0	0	0
Bit 3	0	1	0	1
Bit 2	0	0	0	1
Bit 1	1	0	0	1
Bit 0	1	1	1	1

The second step is to add an even parity to each byte and add a LRC at the end.

	Byte 0	Byte 1	Byte 2	Byte 3	LRC
Parity	1	0	1	0	0
Bit 7	0	0	0	0	0
Bit 6	1	1	1	1	0
Bit 5	0	1	1	1	1
Bit 4	0	0	0	0	0
Bit 3	0	1	0	1	0
Bit 2	0	0	0	1	1
Bit 1	1	0	0	1	0
Bit 0	1	1	1	1	0

Notice that the even parity is the exclusive OR of each bit in the vertical column and

the LRC is the exclusive OR of each bit in the horizontal row. The parity bit for the LRC (or the LRC bit for the parity) will be the exclusive OR of all the data bits.

Now, if any one bit in this 9-row by 5-column matrix is flipped, we can determine which byte is in error by the parity and which bit is in error by the LRC. Rather than asking for retransmission, we simply correct the error. These are very simple error correcting codes, but they illustrate that we can send more bits than the minimum and use those extra bits in a creative way to either detect or correct errors.

RS422, RS485, Ethernet, and CAN are high-speed communication channels. This means the bandwidth and slew rate on the signals are higher than RS232. There is a correspondence between rise time ( $t$ ) of a digital signal and equivalent sinusoidal frequency ( $f$ ). The derivative of  $A \cdot \sin(2\pi ft)$  is  $2\pi f \cdot A \cdot \cos(2\pi ft)$ . The maximum slew rate of this sinusoid is  $2\pi f \cdot A$ . Approximating the slew rate as  $A/t$ , we get a correspondence between  $f$  and  $t$

$$f = 1 / t$$

For example, if the rise time is 5 ns, the equivalent frequency is 200 MHz. Notice that this equivalent frequency is independent of baud rate. So even at 1000 bits/sec, if the rise time is 5 ns, then the signal has a strong 200 MHz frequency component! To deal with this issue, the RS232 protocol limits the slew rate to a maximum of 30V/ $\mu$ s. This means it will take about 400 ns for a signal to rise from -6 to +6 V. Consequently, RS232 signals have frequency components less than 2 MHz. However, to transmit faster than RS232, the protocol must have faster rise times. Electrical signals travel at about 0.6 to 0.9 times the speed of light. This velocity factor (VF) is a property of the cable. For example, VF for RG-6/U coax cable is 0.75, whereas VF is only 0.66 for RG-58/U coax cable. Using the slower 0.66 estimate, the speed is  $v = 2 \cdot 10^8$  m/s. According to wave theory, the wavelength is  $l = v/f$ . Estimating the frequency from rise time, we get

$$l = v * t$$

In our example, a rise time of 5 ns is equivalent to a wavelength of about 1 m. As a rule of thumb, we will consider the channel as a *transmission line* if the length of the wire is greater than  $l/4$ . Another requirement is for the diameter of the wire to be much smaller than the wavelength. In a transmission line, the signals travel down the wires as waves according to the wave equation. Analysis of the wave equation is outside the scope of this book. However, you need to know that when a wave meets a change in impedance, some of the energy will transmit (a good thing) and some of the energy will reflect (a bad thing). Reflections are essentially noise on the signal, and if large enough, they will cause bit errors in transmission. We can reduce the change in impedance by placing terminating resistors on both ends of a long high-speed cable, which are needed for both CAN and Ethernet. These resistors reduce reflections; hence they improve signal to noise ratio.

### 9.1.3. Wireless Communication

The details of exactly how wireless communication operates are beyond the scope of this book. Nevertheless, the interfacing techniques presented in this book are sufficient to implement wireless communication by selecting a wireless module and interfacing it to the microcontroller. In general, one considers bandwidth, distance, topology and security when designing a wireless link. Bandwidth is the fundamental performance measure for a communication system. In this book, we define bandwidth of the system as the information transfer rate. However, when characterizing the physical channel, bandwidth can have many definitions. In general, the bandwidth of a channel is the range of frequencies passed by the channel (Communication Networks by Leon-Garcia). Let  $G_x(f)$  be the gain versus frequency of the channel. When considering EM fields transmitted across space, we can define **absolute bandwidth** as the frequency interval that contains all of the signal's frequencies. **Half-power bandwidth** is the interval between frequencies at which  $G_x(f)$  has dropped to half power (-3dB). Let  $f_c$  be the carrier frequency, and  $P_x$  be the total signal power over all frequencies. The **equivalent rectangular bandwidth** is  $P_x/G_x(f_c)$ . The **null-to-null bandwidth** is the frequency interval between first two nulls of  $G_x(f)$ . The FCC defines **fractional power containment bandwidth** as the bandwidth with 0.5% of signal power above and below the band. The **bounded power spectral density** is the band defined so that everywhere outside  $G_x(f)$  must have fallen to a given level. The purpose of this list is to demonstrate to the reader that, when quoting performance data, we must give both definition of the parameter and the data. If we know the channel bandwidth  $W$  in Hz and the  $SNR$ , we can use the **Shannon–Hartley Channel Capacity Theorem** to estimate the maximum data transfer rate  $C$  in bits/s:

$$C = W * \log_2(1 + SNR)$$

For example, consider a telephone line with a bandwidth  $W$  of 3.4 kHz and  $SNR$  of 38 dB. The dimensionless  $SNR = 10^{(38/10)} = 6310$ . Using the Channel Capacity Theorem, we calculate  $C = 3.4 \text{ kHz} * \log_2(1 + 6310) = 43 \text{ kbits/s}$ .

### 9.1.4. Radio

Figure 9.6 shows a rough image of various electromagnetic waves that exist from radio waves to gamma rays. Visible light constitutes a very small fraction, ranging from 430–770 THz. Bluetooth, ZigBee, and WiFi use an even narrower range from 2.40 to 2.48 GHz, which exists in the microwave spectrum.



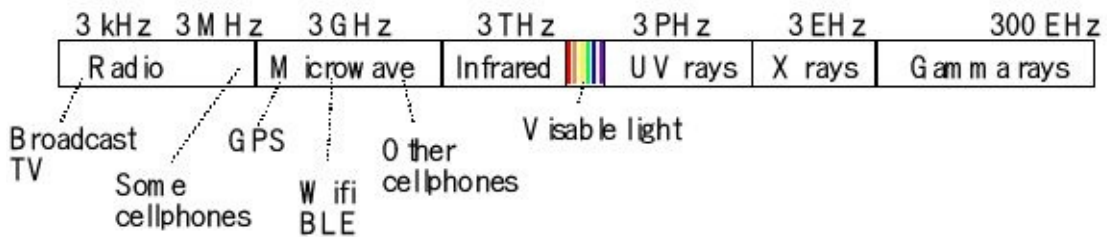


Figure 9.6. Bluetooth communication occurs in the microwave band at about 2.4 GHz.

Table 9.1 shows some general descriptions of the three major communication standards operating in this 2.4 GHz band.

Standard	Description
WiFi	Up to 600 Mbits/sec Fixed wide frequency channels Requires lots of power Support for 2.4 and 5 GHz channels Extensive security features
Bluetooth/BLE	Very low power BT up to 2 Mbps Massive deployed base Frequency hopping Good performance in congested/noisy environment Ease of use, no roaming
ZigBee	Very low power Fixed channels Complex mesh network 250 kbps bandwidth

Table 9.1. Comparison between Wi-Fi, Bluetooth, and ZigBee.

Bluetooth LE could use any of the 40 narrow bands (LL 0 to 39) at 2.4 GHz; these bands are drawn as bumps in Figure 9.7. This figure also shows the WiFi channels, which exist as three wide bands of frequencies, called channel 1, 6 and 11. Because BLE coexists with regular Bluetooth and WiFi, BLE will avoid the frequencies used by other communication devices. LL channels 37, 38 and 39 are used to advertise, and LL channels 9-10, 21-23 and 33-36 are used for BLE communication. BLE has good performance in congested/noisy environments because it can **hop** from one frequency to another. **Frequency Hopping Spread Spectrum (FHSS)** rapidly switches the carrier among many frequency channels, using a pseudorandom sequence known to both transmitter and receiver. This way, interference will only affect some but not all communication.

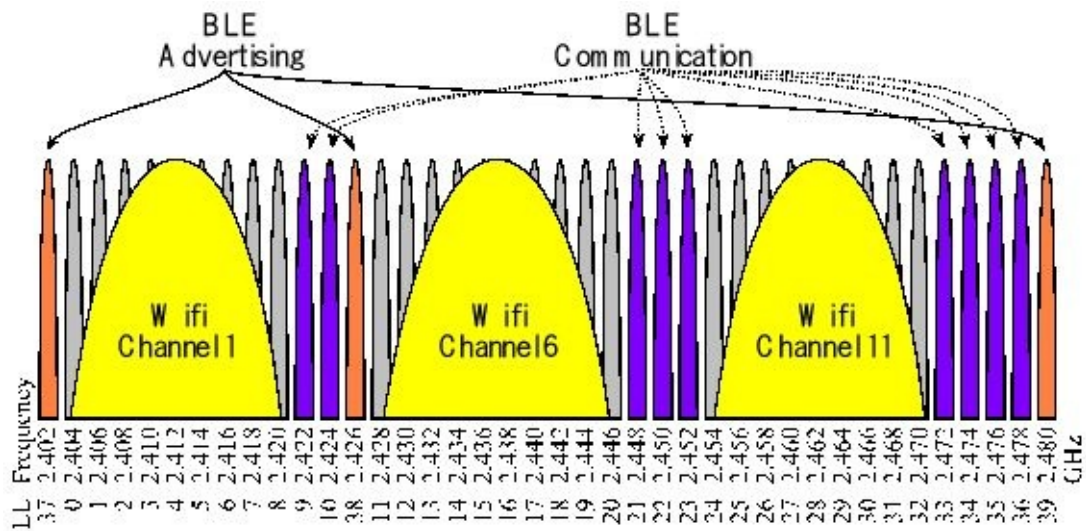


Figure 9.7. The 2.4 GHz spectrum is divided into 40 narrow bands, numbered LL 0 to 39. Each band is  $\pm 1$  MHz.

Figure 9.8 illustrated the inverted F shape of the 2.4 GHz antenna used on the CC2650 LaunchPad. For more information on antenna layout, see <http://www.ti.com/lit/an/swra351a/swra351a.pdf>

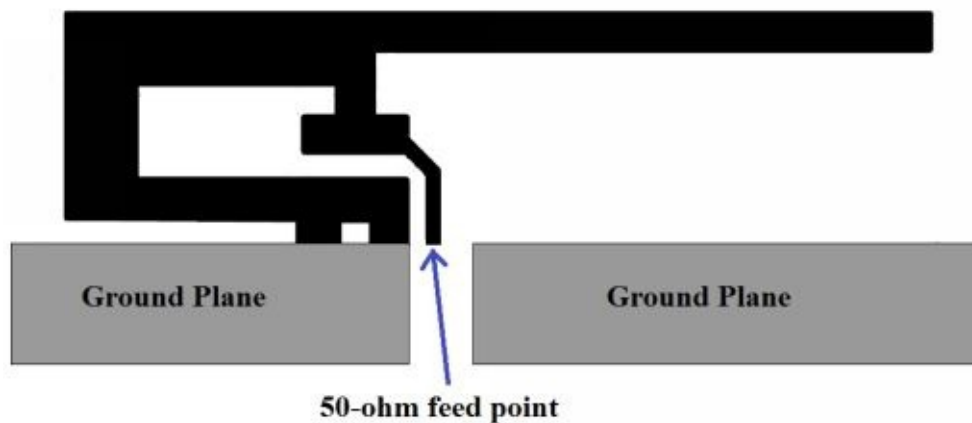


Figure 9.8. One possible layout of the 2.4 GHz antenna.

## 9.2. Controller Area Network (CAN)

### 9.2.1. The Fundamentals of CAN

In this section, we will design and implement a Controller Area Network (CAN). CAN is a high-integrity serial data communications bus that is used for real-time applications. It can operate at data rates of up to 1 Mbits/second, having excellent error detection and confinement capabilities. The CAN was originally developed by Robert Bosch for use in automobiles, and is now extensively used in industrial automation and control applications. The CAN protocol has been developed into an international standard for serial data communication, specifically the ISO 11899.

Figure 9.9 shows the block diagram of a CAN system, which can have up to 112 nodes. There are four components of a CAN system. The first part is the CAN bus consisting of two wires (CANH, CANL) with 120-Ω termination resistors on each end. The second part is the Transceiver, which handles the voltage levels and interfacing the separate receive (Rx) and transmit (Tx) signals onto the CAN bus. The third part is the CAN controller, which is hardware built into the microcontroller, and it handles message timing, priority, error detection, and retransmission. The last part is software that handles the high-level functions of generating data to transmit and processing data received from other nodes.

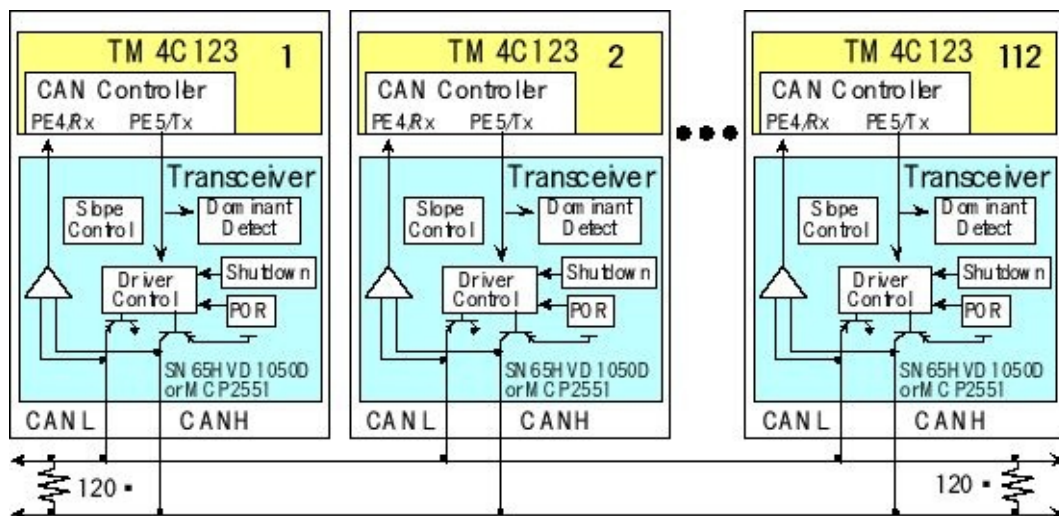
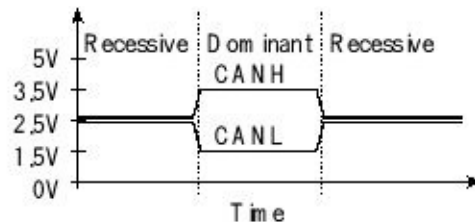


Figure 9.9. Block Diagram of a TM4C-Based CAN communication system

Each node consists of a microcontroller (with an internal CAN controller), and a transceiver that interfaces the CAN controller to the CAN bus. A **transceiver** is a device capable of transmitting and receiving on the same channel. The CAN is based on the “broadcast communication mechanism”, which follows a message-based transmission protocol rather than an address-based protocol. The CAN provides two

communication services: the sending of a message (data frame transmission) and the requesting of a message (remote transmission request). All other services such as error signaling, automatic retransmission of erroneous frames are user-transparent, which implies that the CAN interface automatically performs these functions. Both the TM4C123 and the TM4C1294 have two CAN devices. However, the MSP432 does not have a CAN interface. The physical channel consists of two wires containing in differential mode one digital logic bit. Because multiple outputs are connected together, there must be a mechanism to resolve simultaneous requests for transmission. In a manner similar to open collector logic, there are **dominant** and **recessive** states on the transmitter, as shown in Figure 9.10. The outputs follow a wired-and mechanism in such a way that if one or more nodes are sending a dominant state, it will override any nodes attempting to send a recessive state.



*Figure 9.10. Voltage specifications for the recessive and dominant states.*

**Checkpoint 9.5:** What are the dominant and recessive states in open collector logic?

The CAN transceiver is a high-speed, fault-tolerant device that serves as the interface between a CAN protocol controller (located in the microcontroller) and the physical bus. The transceiver is capable of driving the large current needed for the CAN bus and has electrical protection against defective stations. Typically, each CAN node must have a device to convert the digital signals generated by a CAN controller to signals suitable for transmission over the bus cabling. The transceiver also provides a buffer between the CAN controller and the high-voltage spikes that can be generated on the CAN bus by outside sources. Examples of CAN transceiver chips include the Texas Instruments SN65HVD1050D, AMIS-30660 high speed CAN transceiver, ST Microelectronics L9615 transceiver, Philips Semiconductors AN96116 transceiver, and the Microchip MCP2551 transceiver. These transceivers have similar characteristics and would be equally suitable for implementing a CAN system.

In a CAN system, messages are identified by their contents rather than by addresses. Each message sent on the bus has a unique identifier, which defines both the content and the priority of the message. This feature is especially important when several stations compete for bus access, a process called **bus arbitration**. As a result of the content-oriented addressing scheme, a high degree of system and configuration flexibility is achieved. It is easy to add stations to an existing CAN network.

Four message types or frames can be sent on a CAN bus. These include the **Data Frame**, the **Remote Frame**, the **Error Frame**, and the **Overload Frame**. This

section will focus on the Data Frame, where the parts in standard format are shown in Figure 9.11. The **Arbitration Field** determines the priority of the message when two or more nodes are contending for the bus. For the Standard CAN 2.0A, it consists of an 11-bit identifier. For the Extended CAN 2.0B, there is a 29-bit Identifier. The identifier defines the type of data. The **Control Field** contains the DLC, which specifies the number of data bytes. The **Data Field** contains zero to eight bytes of data. The **CRC Field** contains a 15-bit checksum used for error detection. Any CAN controller that has been able to correctly receive this message sends an Acknowledgement bit at the end of each message. This bit is stored in the Acknowledge slot in the CAN data frame. The transmitter checks for the presence of this bit and if no acknowledge is received, the message is retransmitted. To transmit a message, the software must set the 11-bit Identifier, set the 4-bit DLC, and give the 0 to 8 bytes of data. The receivers can define filters on the identifier field, so only certain message types will be accepted. When a message is received the software can read the identifier, length, and data.

The **Intermission Frame Space** (IFS) separates one frame from the next. There are two factors that affect the number of bits in a CAN message frame. The ID (11 or 29 bits) and the Data fields (0, 8, 16, 24, 32, 40, 48, 56, or 64 bits) have variable length. The remaining components (36 bits) of the frame have fixed length including SOF (1), RTR (1), IDE/r1 (1), r0 (1), DLC (4), CRC (15), and ACK/EOF/intermission (13). For example, a Standard CAN 2.0A frame with two data bytes has  $11+16+36 = 63$  bits. Similarly, an Extended CAN 2.0B frame with four data bytes has  $29+32+36 = 97$  bits.

If a long sequence of 0's or a long sequence of 1's is being transferred, the data line will be devoid of edges that the receiver needs to synchronize its clock to the transmitter. In this case, measures must be taken to ensure that the maximum permissible interval between two signal edges is not exceeded. **Bit Stuffing** can be utilized by inserting a complementary bit after five bits of equal value. Some CAN systems add stuff bits, where the number of stuff bits depends on the data transmitted. Assuming  $n$  is the number of data bytes (0 to 8), CAN 2.0A may add  $3+n$  stuff bits and a CAN 2.0B may add  $5+n$  stuff bits. Of course, the receiver has to un-stuff these bits to obtain the original data.

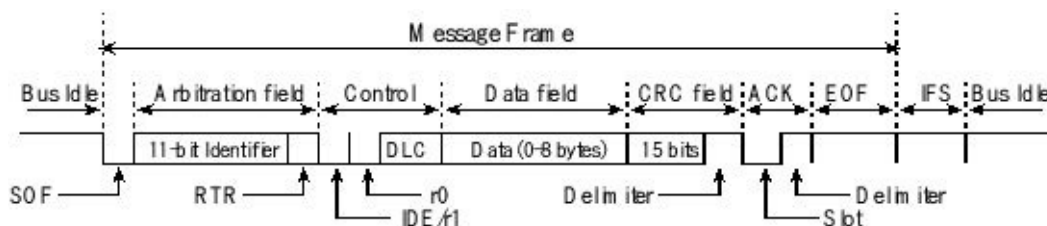


Figure 9.11. CAN Standard Format Data Frame.

The urgency of messages to be transmitted over the CAN network can vary greatly in a real-time system. Typically, there are one or two activities that require high

transmission rates or quick responses. Both bandwidth and response time are affected by message priority. Low priority messages may have to wait for the bus to be idle. There are two priorities occurring as the CANs transmit messages. The first priority is the 11-bit identifier, which is used by all the CAN controllers wishing to transmit a message on the bus. Message identifiers are specified during system design and cannot be altered dynamically. The 11-bit identifier with the lowest binary number has the highest priority. In order to resolve a bus access conflict, each node in the network observes the bus level bit by bit, a process known as bit-wise arbitration. In accordance with the wired-and-mechanism, the dominant state overwrites the recessive state. All nodes with recessive transmission but dominant observation immediately lose the competition for bus access and become receivers of the message with the higher priority. They do not attempt transmission until the bus is available again. Transmission requests are hence handled according to their importance for the system as a whole. The second priority occurs locally, within each CAN node. When a node has multiple messages ready to be sent, it will send the highest priority messages first.

**Observation:** It is confusing when designing systems that use a sophisticated I/O interface like the CAN to understand the difference between those activities automatically handled by the CAN hardware module and those activities your software must perform. The solution to this problem is to look at software examples to see exactly the kinds of tasks your software must perform.

## 9.2.2. Texas Instruments TM4C CAN

A device driver for the CAN network is divided into three components: initialization, transmission, and reception. There is a CAN driver available in TivaWare®. In this section, we will use this driver to develop a simple system that exchanges 4-byte messages between two microcontrollers. Each node generates an interrupt when they receive a CAN message, and the interrupt handler dumps the data either into a mailbox. In this example, the transmission doesn't block, just returns a failure if it can't put, so it will not block or spin. This example was written using the TivaWare® **driverlib** library. Figure 9.12 shows the data flow. There are two IDs used in this example:

```
#define RCV_ID 2
#define XMT_ID 4
```

The CAN ID numbers must be reversed on the other microcontroller. Otherwise, the software functions on the two nodes are identical.

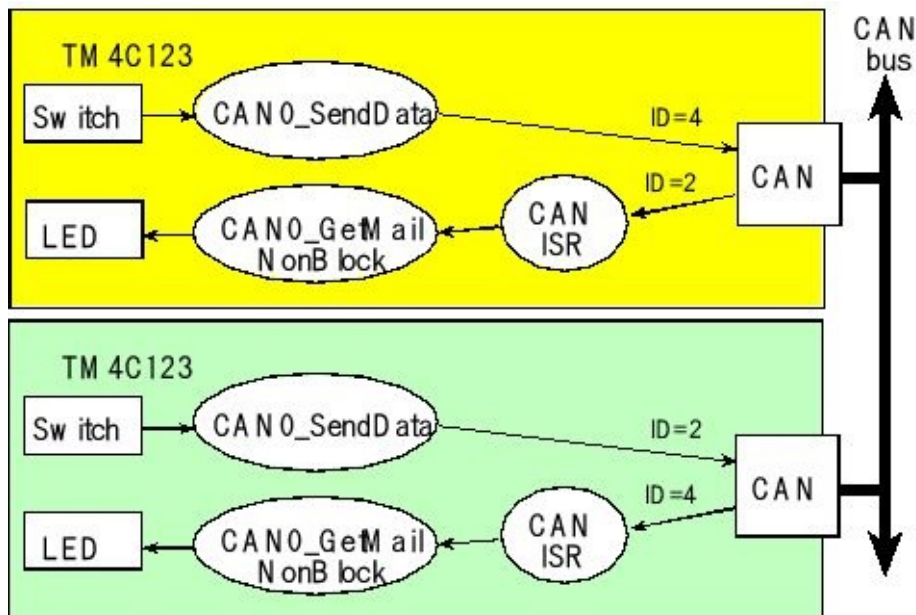


Figure 9.12. Data flow for a simple CAN network.

Transmission uses busy-wait synchronization. However, receiving messages is interrupt driven, and data is passed from the ISR to the user application using a simple mailbox:

```
uint8_t static RCVDData[4];
int static MailFlag; // set when new data arrives
```

The TM4C CAN receiver supports up to 32 message objects. Each message to be sent occupies a unique message object in the 32-object memory of the CAN controller and each receive object matches one of the transmit objects, just on the opposite board. Although this example has only two message objects it could easily be extended to up to 32 objects, but not beyond 32. In this code there are two message objects; the transmission object on one is connected to a receive object on the other. The following helper function sets up one of these 32 message objects, which can be a TX object or an RX object type.

```
void static CAN0_Setup_Message_Object( uint32_t MessageID,
uint32_t MessageFlags, uint32_t MessageLength,
uint8_t * MessageData, uint32_t ObjectID, tMsgObjType eMsgType){
tCANMsgObject xTempObject;
xTempObject.ulMsgID = MessageID; // 11 or 29 bit ID
xTempObject.ulMsgLen = MessageLength;
xTempObject.pucMsgData = MessageData;
xTempObject.ulFlags = MessageFlags;
CANMessageSet(CAN0_BASE, ObjectID, &xTempObject, eMsgType);}
```

The initialization software first configures Port E bits 4,5 to be CAN0. From Table 1.4 we see PE4 is CAN0Rx, and PE5 is CAN0Tx. Next, it initializes the baudrate to 1,000,000 bps. It arms CAN interrupts on error and status change. A status change will occur when an incoming frame is successively received.



The **CAN0\_Setup\_Message\_Object** function will configure one of the 32 message objects. Basically, it will set a filter to allow receive frames with this **RCV\_ID** ID. An interrupt will be generated when receiving this type of frame, but other CAN traffic will be ignored. This function also specifies the expected size in bytes of the payload. Lastly, the CAN module is armed in the NVIC. Interrupts will be enabled in the main program after all devices are initialized.

```
void CAN0_Open(void){uint32_t volatile delay;
MailFlag = false;
SYSCTL_RCGCCAN_R |= 0x00000001; // CAN0 enable bit 0
SYSCTL_RCGCGPIO_R |= 0x00000010; // PortE enable bit 4
for(delay=0; delay<10; delay++){
GPIO_PORTE_AFSEL_R |= 0x30; //PORTE AFSEL bits 5,4
GPIO_PORTE_PCTL_R = (GPIO_PORTE_PCTL_R&0xFF00FFFF)|0x00880000;
GPIO_PORTE_DEN_R |= 0x30;
GPIO_PORTE_DIR_R |= 0x20;
CANInit(CAN0_BASE);
CANBitRateSet(CAN0_BASE, 8000000, CAN_BITRATE);
CANEnable(CAN0_BASE);

CANIntEnable(CAN0_BASE,CAN_INT_MASTER|CAN_INT_ERROR|CAN_INT_STATUS
CAN0_Setup_Message_Object(RCV_ID, MSG_OBJ_RX_INT_ENABLE, 4, NULL,
RCV_ID, MSG_OBJ_TYPE_RX);
NVIC_EN1_R = (1 << (INT_CAN0 - 48)); // IntEnable(INT_CAN0);
return;
}
```

Again, an interrupt is generated when a frame of the appropriate ID is received. The ISR will search the 32 possible message objects for the one that caused the interrupt.

```
void CAN0_Handler(void){uint8_t data[4]; int i;
uint32_t ulIntStatus,ulIDStatus; tCANMsgObject xTempMsgObject;
xTempMsgObject.pucMsgData = data;
ulIntStatus = CANIntStatus(CAN0_BASE, CAN_INT_STS_CAUSE); // cause?
if(ulIntStatus & CAN_INT_INTID_STATUS){ // receive?
ulIDStatus = CANStatusGet(CAN0_BASE, CAN_STS_NEWDAT);
for(i = 0; i < 32; i++){ // test every bit of the mask
if( (0x1 << i) & ulIDStatus){ // if active, get data
CANMessageGet(CAN0_BASE, (i+1), &xTempMsgObject, true);
if(xTempMsgObject.ulMsgID == RCV_ID){
RCVData[0] = data[0]; RCVData[1] = data[1];
RCVData[2] = data[2]; RCVData[3] = data[3];
MailFlag = true; // new mail
}
}
}
}
```



```

}
CANIntClear(CAN0_BASE, ulIntStatus); // acknowledge
}

```

When the user code wishes to receive a message, it calls **CAN0\_GetMailNonBlock**, which is a simple mailbox receiver. This function is nonblocking, meaning if there is no message it returns false. If there is a message, it copies the payload of 4 bytes and returns true. If the RTOS were available, the MailFlag could be replaced with a semaphore. The ISR would signal the semaphore on new data, and the user code could wait on that semaphore.

```

int CAN0_GetMailNonBlock(uint8_t data[4]){
    if(MailFlag){
        data[0] = RCVDData[0];
        data[1] = RCVDData[1];
        data[2] = RCVDData[2];
        data[3] = RCVDData[3];
        MailFlag = false;
        return true;
    }
    return false;
}
int CAN0_CheckMail(void){
    return MailFlag;
}

```

When the user code wishes to transmit data it calls this function, which configures a new message object. This function will send 4 bytes of data to other microcontroller.

```

void CAN0_SendData(uint8_t data[4]){
    CAN0_Setup_Message_Object(XMT_ID,NULL,4,data,XMT_ID,MSG_OBJ_TYPE_TX);
}

```

The **UserTask** ISR periodically reads its switches and creates a transmit object. Because the transmission rate is slower than the network, the transmitter does not wait. It simply creates the message object ( **CAN0\_SendData** ) and schedules it for transmission. When received by the other microcontroller an interrupt is generated and the data is put in a mailbox. The main program on the other microcontroller reads the mail and writes the data out to its LED. Data flows in both directions. Remember to reverse the **XMT\_ID RCV\_ID** values on the two microcontrollers.

```

uint8_t XmtData[4];
uint8_t RcvData[4];
uint32_t RcvCount=0;
uint8_t sequenceNum=0;
void UserTask(void){
    XmtData[0] = PF0<<1; // 0 or 2
    XmtData[1] = PF4>>2; // 0 or 4
    XmtData[2] = 0; // unassigned field
}

```

```

XmtData[3] = sequenceNum; // sequence count
CAN0_SendData(XmtData);
sequenceNum++;
}
int main(void){
    PLL_Init(Bus80MHz);          // bus clock at 80 MHz
    SYSCTL_RCGCGPIO_R |= 0x20;   // activate port F
    while((SYSCTL_PRGPIO_R&0x20) == 0){};
    GPIO_PORTF_LOCK_R = 0x4C4F434B; // unlock GPIO Port F
    GPIO_PORTF_CR_R = 0xFF;       // allow changes to PF4-0
    GPIO_PORTF_DIR_R = 0x0E;      // make PF3-1 output
    GPIO_PORTF_AFSEL_R = 0;       // disable alt funct
    GPIO_PORTF_DEN_R = 0x1F;      // enable digital I/O on PF4-0
    GPIO_PORTF_PUR_R = 0x11;      // enable pullup on inputs
    GPIO_PORTF_PCTL_R = 0x00000000;
    GPIO_PORTF_AMSEL_R = 0;       // disable analog functionality on PF
    CAN0_Open();
    Timer3_Init(&UserTask, 1600000); // initialize timer3 (10 Hz)
    EnableInterrupts();
    while(1){
        if(CAN0_GetMailNonBlock(RcvData)){
            RcvCount++;
            PF1 = RcvData[0];
            PF2 = RcvData[1];
            PF3 = RcvCount; // heartbeat
        }
    }
}
}

```

*Program 9.1. Very simple CAN network example.*

In this simple example, there is just one transmit ID type and one receive ID type, but you could rewrite the transmitter and receiver to allow multiple ID types. In this case, the message ID (11-bit ID) and the object ID (0 to 31) are the same. In general, there could be 2048 IDs, but in this example only the first 32 can be used. The transmit messages are sent without interrupts, but the receive messages will trigger an interrupt. It would take three steps to expand to more receive IDs. First, we would call **CAN0\_Setup\_Message\_Object** multiple times during initialization, once for each type of message we wish to receive (obviously, giving each a unique ID, up to 32). Second, for each possible ID, we would duplicate the **if(xTempMsgObject.ulMsgID==RCV\_ID){}** test in the ISR to check if a desired message has been received. Third, each message ID would need its own mailbox or FIFO. This way the user tasks could be signaled when the appropriate data is available. Expanding the system to support more transmit message IDs is simple. We simply duplicate **CAN0\_SendData** function for each message ID we wish to send.

## 9.3. Embedded Internet

This section provides a brief introduction to the Internet as well as present low-level details of the Ethernet controller on a Tiva microcontroller. For an excellent description of the **TCP/IP** (Transmission Control Protocol/Internet Protocol) protocol the reader is referred to W. Richard Stevens, TCP/IP Illustrated, Volume 1: The Protocols. For a general description of the internet of things, see Vasseur and Dunkels, Interconnecting Smart Objects with IP. These two books provide good overviews of network technologies used for connecting devices.

### 9.3.1. Abstraction

In a manner similar to ZigBee, TCP/IP packets hop from one network to another as they travel from source to destination, see Figure 9.13. The network schedules communication and provides routing from source to destination. Communication channels such as USB and CAN have scheduling mechanisms to guarantee real-time performance. In particular, USB allows for prenegotiated bandwidth, so important data can be sent in real time. Because of the priority, important CAN messages will have bounded latency. TCP/IP although fast and reliable has no built-in guarantees of timing. Nevertheless, the use of TCP/IP is growing in the embedded world. Often TCP/IP is fast enough and reliable enough for embedded applications, even if response time is uncertain.

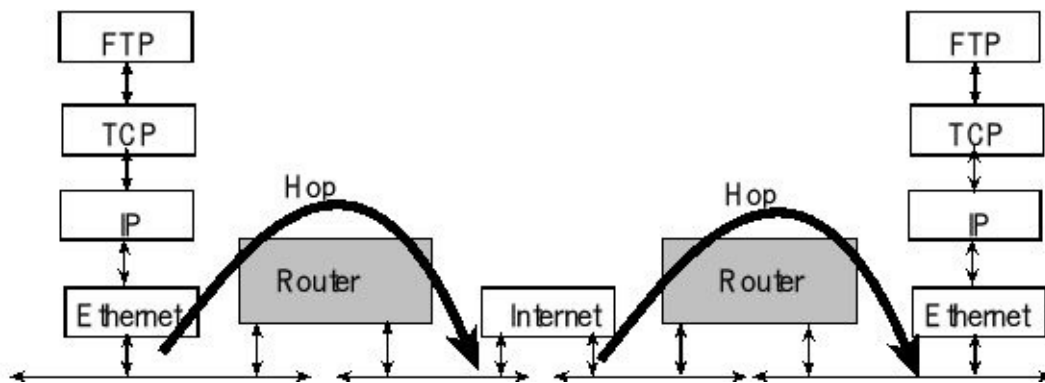


Figure 9.13. Packets on the internet hop from one network to another.

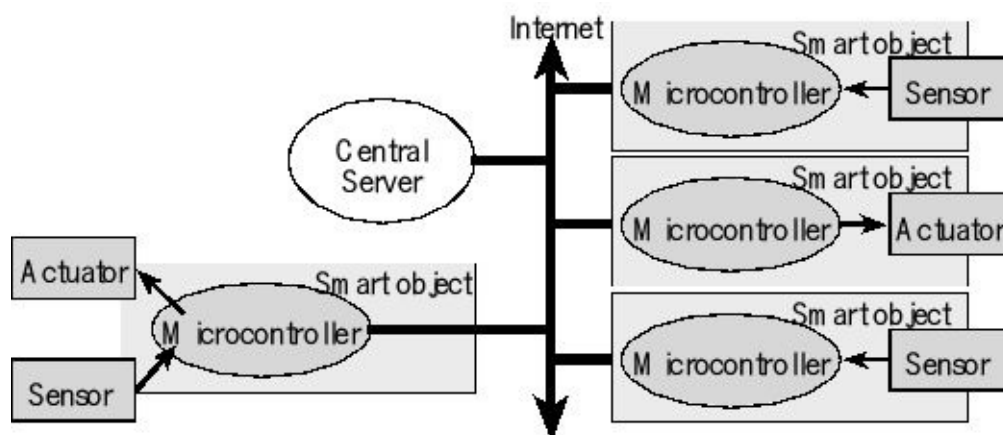
When faced with a complex problem, one could develop a solution on one powerful and **centralized** computer system. Alternatively a **distributed** solution could be employed using multiple computers connected by a network. The processing elements in Figure 9.14 may be a powerful computer, a microcontroller, an ASIC, or a smart sensor/actuator. Another name given for an embedded system connected to the internet is **smart object**. Smart objects include sensors to collect data, processing to

detect events and make decisions, and actuators to manipulate the local environment.

Table 9.2 lists some existing applications and the things they sense or control. There are many reasons to consider a distributed solution (network) over a centralized solution. Often multiple simple microcontrollers can provide a higher performance at lower cost compared to one computer powerful enough to run the entire system. Some embedded applications require input/output activities that are physically distributed. For real-time operation there may not be enough time to allow communication between a remote sensor and a central computer. Another advantage of distributed system is improved debugging. For example, we could use one node in a network to monitor and debug the others. Often, we do not know the level of complexity of our problem at design time. Similarly, over time the complexity may increase or decrease. A distributed system can often be deployed that can be scaled. For example, as the complexity increases more nodes can be added, and if the complexity were to decrease nodes could be removed.

Industrial Automation	Factories, machines, shipping
Environment	Weather, pollution, public safety
Smart Grid	Electric power, energy delivery
Smart Cities	Transportation, hazards, public services
Social Networks	Ideas, politics, sales, and communication
Home Networks	Lighting, heat, security, information
Building Networks	Energy, hazards, security, maintenance
Structural Monitors	Bridges, roads, building
Health Care	Heart function, medical data, remote care
Law enforcement	Crime, public safety

**Table 9.2. Applications of smart objects.**



*Figure 9.14. The internet of things places input, output and processing at multiple locations connected together over the internet.*

The **TCP/IP** model of the **Internet** does not adhere to such a strict layered structure, but does recognize four broad layers: scope of the software application; the end-to-end transport connection; the internetworking range; and the direct links as shown on the right of Figure 9.15. Examples of applications include Telnet, FTP (File Transfer Protocol), and SMTP (Simple Mail Transfer Protocol). Examples of transport include TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). TCP provides reliable, ordered delivery of data from a software task on one computer to another software task running on another computer. For applications that do not require reliable data stream service UDP can be used. UDP provides a datagram service that emphasizes reduced latency over reliability. Examples of network include IP (Internet Protocol), ICMP (Internet Control Message Protocol) and IGMP (Internet Group Management Protocol). **Ethernet** is the physical link explored later in this section. In this section we will develop projects at the application layer. The communication of **bits** happens at the physical layer, **frames** at the data link layer, **packets** or **datagrams** at the network layer, **segments** at the transport layer, and **messages** at the application layer.

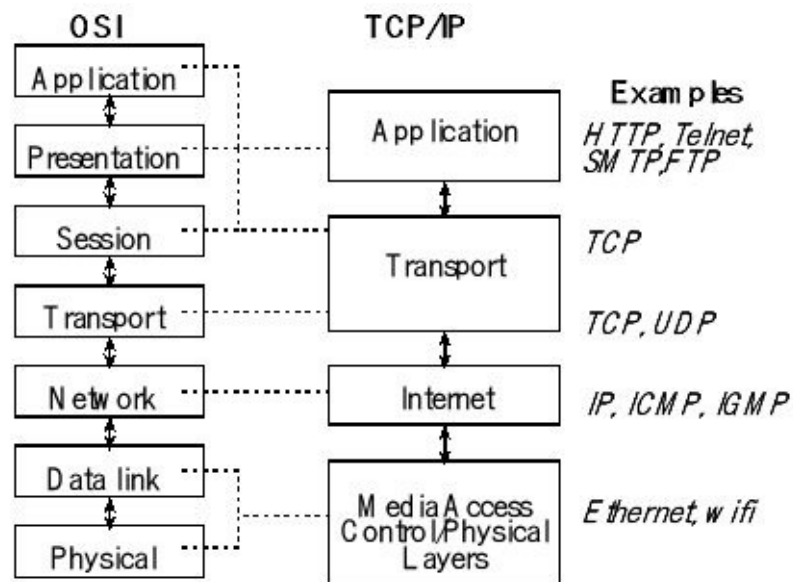


Figure 9.15. The TCP/IP model has four layers.

### 9.3.2. Message Protocols

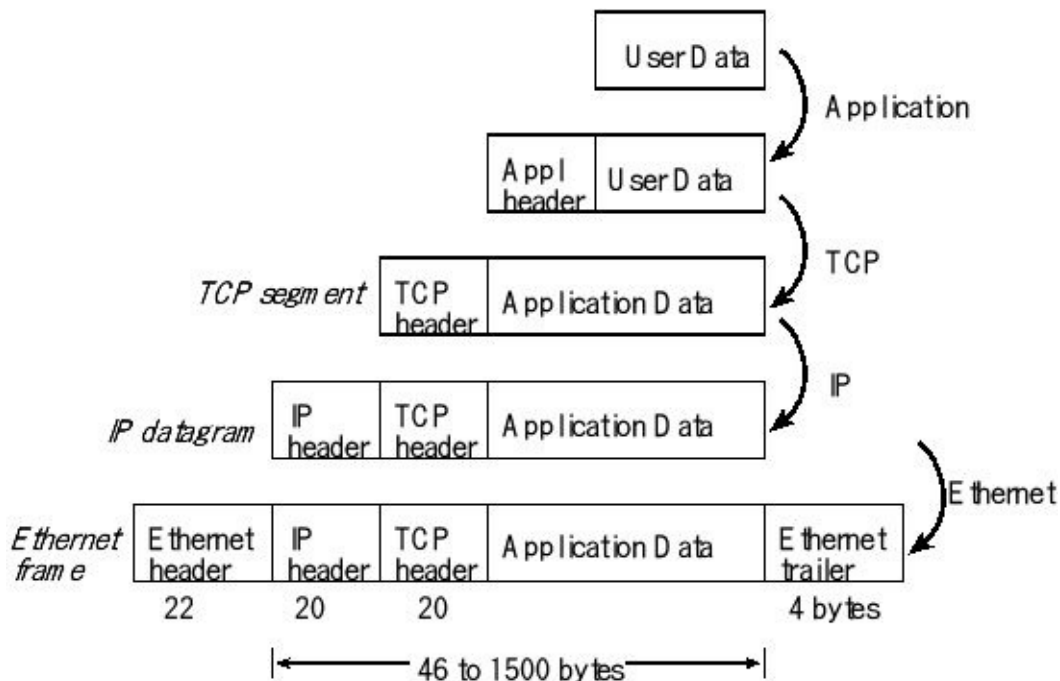
The layered format can be seen in the message packet formats, as overviewed in Figure 9.16. At the lowest level are Ethernet frames, which contain a header, 46 to 1500 bytes of payload, and a trailer. The header includes address, type and length information. If there is less than 46 bytes of Ethernet data, zeros are added (padding) to make the Ethernet payload at least 46 bytes. The trailer includes error checking (CRC). At the IP level, packets include a header and payload. The header of an IP packet includes a 32-bit destination IP address, typically shown as four 8-bit numbers (e.g., 176.31.244.1). Some of these IP addresses are reserved for communicating within nodes on a local network. The Domain Name System (DNS)

host can be used to translate domain names to IP addresses. Computers that communicate only with each other via TCP/IP, but are not connected to the Internet, need not have globally unique IP addresses. IP addresses for private networks are listed in Table 9.3. These IP addresses could be used for systems that use TCP/IP to communicate, but are not connected to the internet.

<i>Start</i>	<i>End</i>	<i>Number of addresses</i>
10.0.0.0	10.255.255.255	$2^{24}$
172.16.0.0	172.31.255.255	$2^{20}$
192.168.0.0	192.168.255.255	$2^{16}$

**Table 9.3. Private IP addresses.**

Because of the growth of the internet, the 32-bit IP address (IPv4) is being replaced with a 128-bit address (IPv6), which will provide for about  $3 \cdot 10^{38}$  addresses.



*Figure 9.16. Overview of message packets used at various layers.*

### 9.3.3. Ethernet Physical Layer

The goal of Ethernet is to provide reliable communication over an unreliable medium. The Ethernet physical layer has evolved over time and includes many physical media interfaces. Ethernet speed ranges over 2 orders of magnitude. The most common forms used are 10BASE-T, 100BASE-TX, and 1000BASE-T. All three utilize twisted pair cables and 8P8C modular connectors. They run at 10 Mbit/s, 100 Mbit/s, and 1 Gbit/s, respectively. Fiber optic variants of Ethernet offer high performance, electrical isolation and distance (tens of kilometers with some

versions). In general, network protocol stack software will work similarly on all varieties. The left side of Figure 9.17 shows two processing elements connected with Ethernet. The transmitter of one element is connected to the receiver of the other. If more than two processing elements are connected to the same physical medium, then collisions could occur. One solution to reduce collisions is to use an Ethernet switch (right side of Figure 9.17).

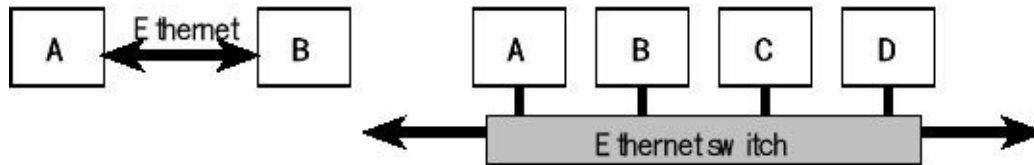


Figure 9.17. Ethernet has a bus-based topology.

Hubs and switches allow multiple devices to exist on the same network. They differ in the way that they pass the network traffic that they receive. A **hub** repeats incoming frames to all nodes on the network. If there are a small number of nodes and the traffic is light, this simple approach is adequate. A **switch** learns the addresses of the nodes connected to it; this way, an incoming frame is sent only to the proper node. If there are a lot of nodes, this selective retransmission provides a significant improvement in performance over a hub. A **router** sits between two networks and passes frames from one network to another, see Figure 9.13.

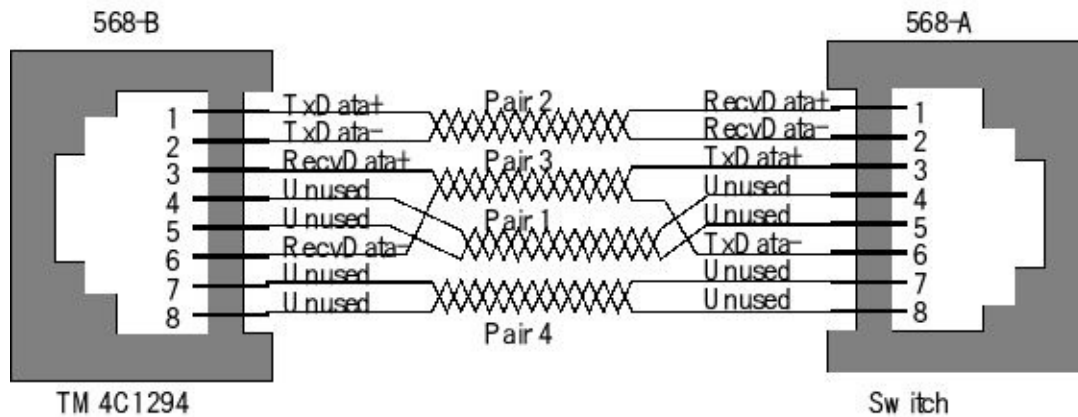
Packets from one element are sent to the appropriate destination. If a collision were to occur (sending packets to the same destination at the same time), then the switch will delay one packet to avoid the collision. From the viewpoint of the nodes, the network looks like a bus-based topology. For example, if processing element A wishes to send a packet to processing element C, it transmits the packet that is addressed to C onto the bus, and the C receives it.

Table 9.4 shows the pin assignments in the 8-wire 568-B connectors. The 568-A connector has the transmit and receive pins reversed. These two connector configurations are similar to the **data terminal equipment** (DTE) and the **data communication equipment** (DCE) of RS232 described in Section 4.9 of Volume 2. When connecting a processing element to a switch, a 568-B connector is used on the processing element and a 568-A connector is used on the switch. This way a straight-through 8-wire cable can be used (Figure 9.18). When connecting two processing element to each other, both elements use 568-B connectors. For this situation the pairs 2 and 3 are reversed in the cable can be used (Figure 9.19 and Table 9.5).

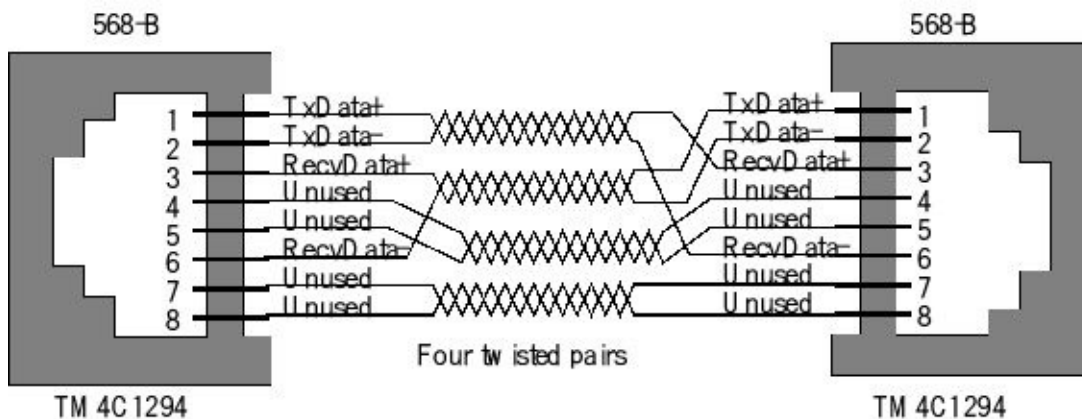
<i>Pin</i>	<i>Color</i>	<i>Pair</i>	<i>Description</i>
1	white/orange	2	TxData +
2	orange	2	TxData -
3	white/green	3	RecvData +
4	blue	1	Unused
5	white/blue	1	Unused
6	green	3	RecvData -

7	white/brown	4	Unused
8	brown	4	Unused

**Table 9.4. Pin assignments on a 568-B Ethernet connector.**



*Figure 9.18. Ethernet cable between a microcontroller and a switch.*



*Figure 9.19. Ethernet cable between two microcontrollers.*

Pin	Left color	Left signal	Cable	Right color	Right signal
1	white/orange	TxData +		white/green	TxData +
2	orange	TxData -		green	TxData -
3	white/green	RecvData +		white/orange	RecvData +
4	blue	Unused		blue	Unused
5	white/blue	Unused		white/blue	Unused
6	green	RecvData -		orange	RecvData -
7	white/brown	Unused		white/brown	Unused
8	brown	Unused		brown	Unused

**Table 9.5. Pin assignments for a crossover Ethernet cable.**



### 9.3.4. Ethernet on the TM4C1294

The Ethernet Controller consists of a fully integrated media access controller (MAC) and network physical (PHY) interface. The Ethernet Controller conforms to IEEE802.3 specifications and fully supports 10BASE-T and 100BASE-TX standards. To fully understand this section, you must read the TM4C1294 datasheet. In other words, this section is meant to supplement, rather than replace the datasheet.

As shown in Figure 9.20, the Ethernet Controller is functionally divided into two layers: the Media Access Controller (MAC) layer and the Network Physical (PHY) layer. These layers correspond to the OSI model layers 2 and 1. The microcontroller accesses the Ethernet Controller via the MAC layer. The MAC layer provides transmit and receive processing for Ethernet frames. The MAC layer also provides the interface to the PHY layer via an internal Media Independent Interface (MII). The PHY layer communicates with the Ethernet bus.

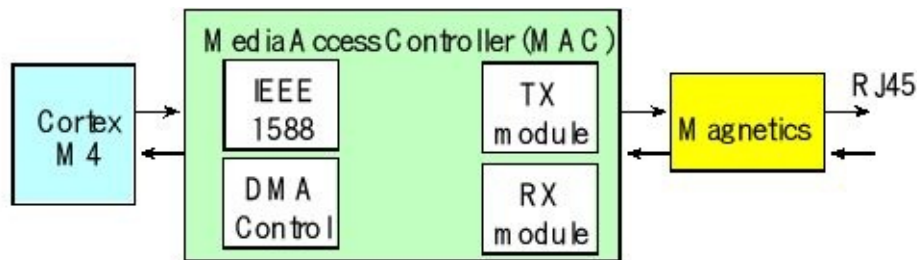
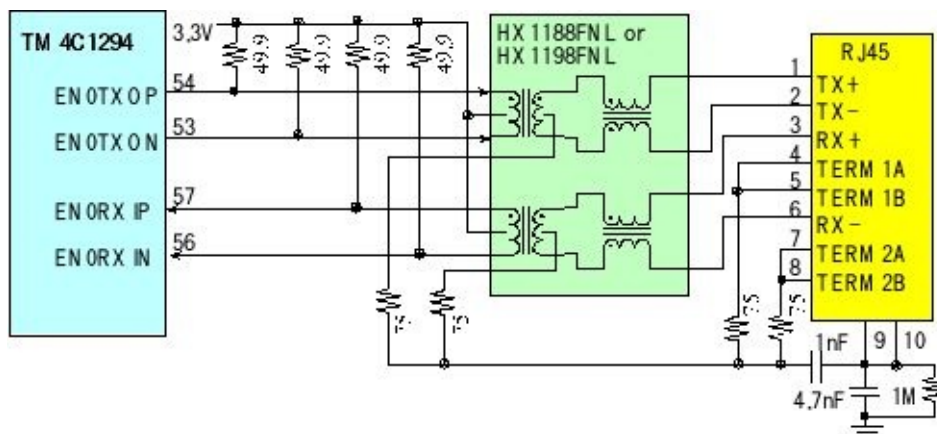


Figure 9.20. The Ethernet port on the microcontroller implements the MAC and PHY layers.

Figure 9.21 shows the hardware interface between the TM4C1294 and the Ethernet cable as it is implemented on the Connected LaunchPad. The data is coupled onto the bus via transform coupling. The transforms are connected to the RJ45 jack. Only four of the eight wires are used and there are no ground pins in the cable. There are two activity LEDs. By default, these pins are configured as GPIO signals (PF3 and PF2). For the PHY layer to drive these signals, they must be reconfigured to their alternate function. When configured for Ethernet operation, LEDs D4 (PF0) and D3 (PF4) on the connected LaunchPad are controlled by the Ethernet MAC to indicate connection and transmit/receive status.



*Figure 9.21. Electrical interface between the microcontroller and the Ethernet cable.*

An Ethernet data packet is called a **frame** (Figure 9.22). A frame begins with preamble and start frame delimiter, followed by an Ethernet header featuring destination and source MAC addresses. Whether the **Length/Type** field is a length or a type depends on the numeric value. If the value of the Length/Type field is less than or equal to 1500 decimal, it indicates the number of MAC client data bytes. If the value of this field is greater than or equal to 1536 decimal, then it is type interpretation. The meaning of this field when the value is between 1500 and 1536 decimal is unspecified. The middle section of the frame consists of **payload** data including any headers for other protocols (e.g., Internet Protocol) carried in the frame. The minimum frame size is 46 bytes. If the frame size is too small, the Ethernet Controller automatically appends extra bytes (a pad) to make it at least 46 bytes. The frame ends with a **frame check sequence** (FCS) is a 32-bit cyclic redundancy check (CRC), which is used to detect corruption of data in transit. The CRC is computed over the destination address, source address, length/type, and data (including pad) fields using the CRC-32 algorithm. For transmitted frames, this field is automatically inserted by the MAC layer, unless disabled by clearing the CRC bit in the MACTCTL register. For received frames, this field is automatically checked. If the FCS does not pass, the frame is not placed in the RX FIFO, unless the FCS check is disabled by clearing the BADCRC bit in the MACRCTL register.



*Figure 9.22. An Ethernet frame can hold 46 to 1500 bytes.*

**Autonegotiation** is the procedure by which two connected devices choose common transmission parameters, such as speed and duplex mode. Autonegotiation was first introduced as an optional feature for 100BASE-TX, but it is also backward compatible with 10BASE-T. Autonegotiation is mandatory for 1000BASE-T.

Example software for this Ethernet link can be found in TI's TivaWare.

- enet\_io                    Ethernet-based I/O Control
- enet\_lwip                Ethernet with a Lightweight TCP/IP stack (lwIP)
- enet\_uip                 Ethernet with uIP TCP/IP Stack
- enet\_weather            Ethernet with lwIP Weather Application

For more information on lwIP, see <http://savannah.nongnu.org/projects/lwip/>

---

## 9.4. Internet of Things

### 9.4.1. Basic Concepts

With the proliferation of embedded systems and the pervasiveness of the internet, it is only natural to connect the two. The internet of things (IoT) is the combination of embedded systems, which have sensors to collect data and actuators to affect the surrounding, and the internet, which provides for ubiquitous remote and secure communication. This section will not describe how the internet works, but rather we will discuss both the general and specific approaches for connecting embedded systems to the internet. (References for internet in general and IoT in specific see: W. Richard Stevens, TCP/IP Illustrated, Volume 1: The Protocols and Vasseur and Dunkels, Interconnecting Smart Objects with IP).

**Challenges.** On a local scale, the design of smart objects faces the same challenges existing in all embedded systems: power, size, reliability, longevity, and cost. Luckily the deployment of billions of microcontrollers into the market has created a technology race to reduce power, size and cost while increasing the performance. At the microcontroller level things are getting smaller, but at the network level, complexity is increasing and protocols are constantly changing as the world's thirst for information and communication rapidly grows.

**Standardization.** The existence of standards allows for a wide variety of objects to communicate with each other. Adhering to a standard will increase the acceptance of our device by customers, and allow our customers to apply our device to solve problems we never envisioned. **uIP** is a light-weight implementation of the IP stack specifically designed to operate with the available memory resources of smart objects. In this section we will start with a microcontroller with the hardware and software to implement TCP/IP protocols, and build our application on top of this standard.

**Interoperability** means our device can function with a wide range of other devices made with different technologies, sold by different vendors, and produced by different companies.

**Evolution** is the process of how new technologies are introduced into the market. If there is one constant in this world, it is that things will change. Every thousand years, one big discovery fundamentally changes how we operate (fire, language, metal tools). More frequently, change is introduced gradually such that those technologies that give us a competitive advantage survive. If we build our business model on the premise evolutionary change, then we can be nimble to deploy new technology when it provides lower cost and/or better performance.

**Stability.** Even though technology will advance, our customers demand products that work reliably, for a long time, and in a manner with which they are comfortable. Over the last 50 years, automotive technology has drastically improved, but the driving experience, how we drive, has remained almost constant.

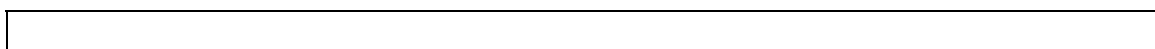
**Abstraction.** You will notice the approach in this section differs widely from the other examples in this book. The rest of the book deploys a bottom up approach. With bottom-up education, the details are first explained, so there is no magic, and then abstraction occurs by encapsulating that we fully understand. In this section we will purchase hardware and software with capabilities to communicate with the internet, and use this abstraction without fully understanding how some of the lower levels operate.

**Scalability.** ARM reports over 50 million devices with an ARM core have been shipped from 1993 to 2013, and predicts another 50 billion before the end of this decade. In order to be effective and profitable, we need to develop systems that can scale.

**Security.** Because embedded systems are deployed in life-critical situations, and because the quality of service affect our profits, we must protect the system from a determined adversary. A chain is only as strong as its weakest link. Security cannot be obtained simply by operating in secret, because once the secret is out, the system will be extremely vulnerable. “Security by obscurity” is a very poor design method. Security involves more than encrypting the data. The first aspect of security is **confidentiality**. We must decide what it means to view/change the data and who has the right to read/write. Authentication is the means to ensure the identity of the sender is correct. Confidentiality will require both logical and physical measures to protect against an attack. Encryption makes it harder for an unauthorized party to view a message. The second aspect is data **integrity**. For most of the applications listed in Table 9.2 it is important that data reach the rightful recipient in an unaltered fashion. To support network integrity, we need techniques that support both detection and prevention. The third aspect is **availability**. A secure communication not only requires the correct data arrive at the correct place, but also at the correct time. A Denial of Service (DoS) attack attempts to breach the availability of the network. For wired networks, we can reroute traffic along multiple paths. With wireless networks, we can channel hop by switching channels on a pseudorandom fashion, making it harder for an attacker to jam. For more information on security, see Frank Stajano, [Security for Ubiquitous Computing](#).

## 9.4.2. UDP and TCP Packets

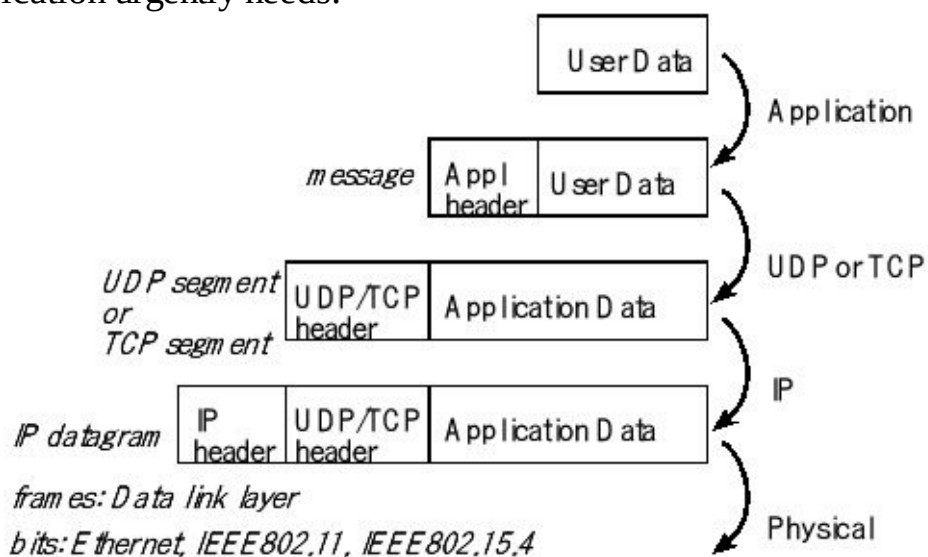
The UDP header is 8 bytes and contains the source port, destination port, length, and checksum, see Table 9.6 and Figure 9.23. The IP address specifies the node, and ports are addresses within the source and destination nodes.



Source port: 16-bit number of the process that sent the packet, could be zero
Destination port: 16-bit number of the process to receive the packet.
Length: 16-bit number specifying the size in bytes of the data to follow
Checksum: 16-bit modulo addition of all data, UDP header, and IP header

**Table 9.6. UDP header format.**

The TCP header is 20 bytes with the possibility of additional and optional information, see Table 9.7. The sequence and acknowledgment numbers allow the receiver to properly sort segments of data that were received out of order. The flags specify different modes of the TCP communication. The SYN flag means the first of a sequence of packets, and the FIN flag means the last. The RST flag terminates a connection. The URG flag means the urgent pointer specifies a piece of data the application urgently needs.



*Figure 9.23. Overview of message packets used at various layers.*

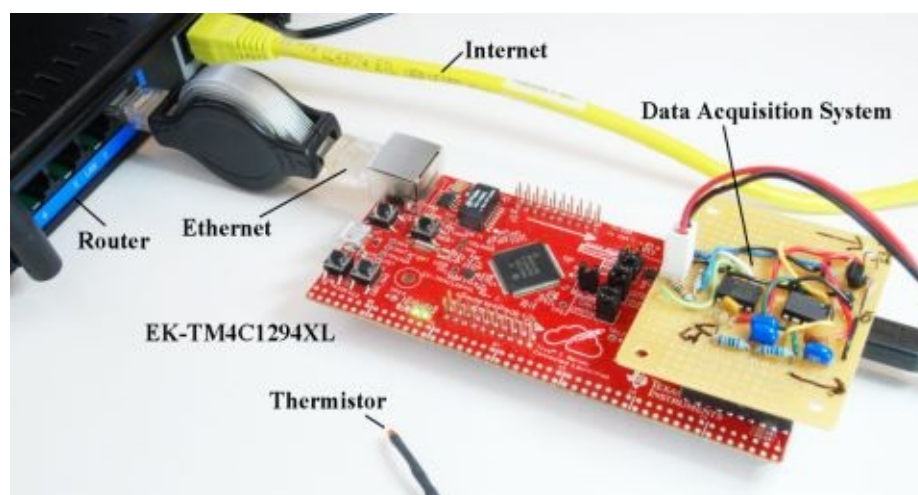
Source port: 16-bit number of the process that sent the packet, could be zero
Destination port: 16-bit number of the process to receive the packet.
Sequence number: 32-bit number defining the position of this data
Acknowledgement: 32-bit number of the next data expected to be received

Hlen: 4-bit field of the header size (including options) divided by 4
Flags: 6-bit field with FIN, SYN, RST, PSH, ACK, and URG
Window: 16-bit number specifying the number of bytes the receiver can accept
Checksum: 16-bit modulo addition of all data, TCP header, and IP header
Urgent pointer: 16-bit field pointing to a place in the stream urgently needed

**Table 9.7. TCP header format.**

### 9.4.3. Web server

This first application creates a web server that maintains a web page displaying local data, see Figures 9.24 and 9.25. The components of the system are a sensor and sensor interface, an EK-TM4C1294XL LaunchPad, Texas Instruments TivaWare, and a router connected to the Internet. The Dynamic Host Configuration Protocol server provides an IP address, and is typically initiated via a DHCP broadcast, when it connects. DHCP provided the address 192.168.0.107, a local address on its network. This example was built on top of the uIP stack delivered as part of TivaWare. First, you need to download TivaWare. I first ran the `enet_uip` example found in the `TivaWare_C_Series-2.1.0.12573\examples\boards\ek-tm4c1294xl\enet_uip` folder. I copied this example, and changed the web server as shown in Program 9.2.



*Figure 9.24. The thermistor measures temperature and the LaunchPad serves pages to the internet.*

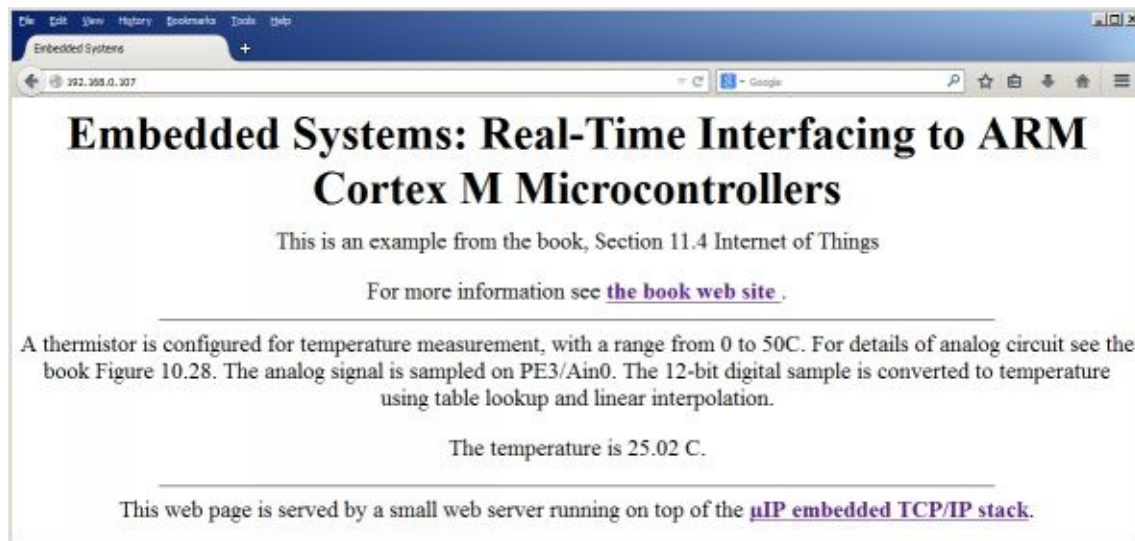


Figure 9.25. The thermistor measures temperature and the LaunchPad serves pages to the internet.

Program 9.2 shows the code you need to modify to create your own remote sensor smart object. When another node sends a request to this server, this node will respond with html code to render the page. The page is divided into three parts. The first part (`default_page_buf1of3`) and last part (`default_page_buf3of3`) are fixed. The application callback function, `httpd_appcall`, is invoked when the web page is requested. This callback function calls our application function `Board_Update` which collects sensor data from the thermistor and rebuilds the middle part of the html code (`default_page_buf2of3`). The meta code automatically refreshes every 5 seconds.

```
const char default_page_buf1of3[] =
"HTTP/1.0 200 OK\r\n"
"Server: UIP/1.0 (http://www.sics.se/~adam/uiip/)\r\n"
"Content-type: text/html\r\n\r\n"
"<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">"
"<html> <head>"
"<meta http-equiv='refresh' content='5'>"
"<title>Embedded Systems</title></head>"
"<body> <center>"
"<h1>Embedded Systems: Real-Time Interfacing"
"to ARM Cortex M Microcontrollers</h1>"
"<p>This is an example from the book, Section 11.4 Internet of Things</p>"
"<p> For more information see "
"<a href='\"http://users.ece.utexas.edu/~valvano/arm/outline.htm\"'">"
"<b>the book web site</b> </a>."
"<hr width='75%'>"
"<p>A thermistor is configured for temperature measurement, "
"with a range from 0 to 50C. "
"For details of analog circuit see the book Figure 9.21. "
"The analog signal is sampled on PE3/Ain0. "
"The 12-bit digital sample is converted to temperature using table lookup "
"and linear interpolation.</p> "
"<p>The temperature is ";
uint32_t const buf1of3_Size = (sizeof(default_page_buf1of3) - 1);
char default_page_buf2of3[] = "12.01";
uint32_t buf2of3_Size = (sizeof(default_page_buf2of3) - 1);
```

```

const char default_page_buf3of3[] =
" C.</p>"
"<hr width=\"75%\">"
"<p>This web page is served by a small web server running on top of "
"the <a href=\"http://www.sics.se/~adam/uiip/\"><b>&micro;IP embedded TCP/IP "
"stack</b></a>.</center> </body> </html>";
uint32_t const buf3of3_Size = (sizeof(default_page_buf3of3) - 1);
void Board_Update(void){uint32_t data,temperature;
data = ADC0_InSeq3(); // 12-bit ADC, 0 to 4095
temperature = ADC2Temperature(data); // temperature, 0.01C
Fix2Str(temperature,default_page_buf2of3); // 5 ASCII characters
buf2of3_Size = 5; // in this case it is fixed size (but it could vary)
}

```

*Program 9.2. The thermistor measures temperature and the LaunchPad serves pages to the internet.*

To run the internet examples described in this section download and unzip the IoT examples into **examples\boards** so the directory path looks like this

```

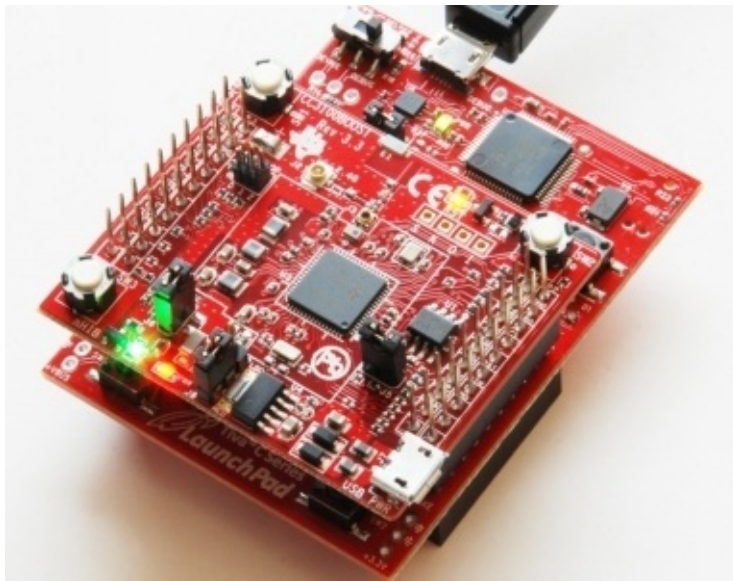
TivaWare_C_Series-2.1.0.12573
  examples
    boards
      CC31xxxx
        ek-tm4c1294xl-enet_uiip_temperature
        ek-tm4c123gxl-boost-cc3100_basic_wifi_UDP
        ek-tm4c123gxl-boost-cc3100_starter
        ek-tm4c1294xl-boost-cc3100_starter

```

#### 9.4.4. UDP communication over WiFi

The approach for implementing a smart object over WiFi is to begin with a hardware/software platform that implements IEEE801.11 WiFi. The CC3100BOOST is a BoosterPack that can be used with the MSP430 LaunchPad, the TM4C123 LaunchPad, the TM4C1294 LaunchPad, or with a CC31XXEMUBOOST emulation module, see Figure 9.26. The emulation module can be used early in a project to develop wireless applications using a “generic” microcontroller. After a prototype is configured, the project can select a microcontroller and design the actual smart object. In this design we will use either of the two TM4C LaunchPads and develop a solution that transmits UDP packets from one smart object to another. UDP is simpler than TCP and appropriate for applications requiring simplicity and speed. Furthermore, to use UDP the application must tolerate lost or out of order packets. UDP provides a best-effort datagram delivery service.





*Figure 9.26. The CC3100 booster pack provides IEEE802.11 wireless connectivity.*

The actual TCP/IP software stack resides in firmware on the booster pack itself. Therefore, when using any of the wireless booster packs the first step is to upgrade the firmware. One way to upgrade the firmware is to use the CC31XXEMUBOOST emulation module. The examples of this section ran on version 3.3 booster packs without needing to upgrade the firmware.

Program 9.3 shows the client software, which samples the ADC and sends UDP packets. Line 1 specifies the name of the access point (AP) to which the node will connect. There is a mechanism using SmartConfig to automate this discovery, but in this example I named the AP **Valvano** so I used a manual method to define the connection between the node and AP. The UDP payload will have a type field, which is defined in line 2. The destination IP address is hard-coded in line 3. For this application, the server was at IP address at 192.168.0.101, which in hex is C0.A8.00.65. The port number, which is a 16-bit value defining which process in the server should receive the data, is specified in line 4. There are a long list of registered port numbers that have special purposes, so I chose a port number larger than 1024 to avoid selecting any of these special purpose port numbers. Lines 5 and 6 define the payload for the UDP packet. Line 15 sets the bus clock to 50 MHz. The PLL needs to be active for the ADC to operate. Line 16 initializes the ADC channel 7 using PD0. Line 17 initializes the CC3100. After executing line 18 we will be connected and have IP address. Line 19 will return the network configuration. Lines 21-24 define the address and port to which the USP packet will be sent. Line 25 defines and opens a socket. In this example we leave the socket open, but it is ok to close the socket, go into low-power mode, and reopen the connection after sleeping. Lines 26-29 will sample the ADC and create a new message. Line 30 sends the UDP packet through the open socket. The wait in line 32 defines the rate at which packets are sent. Each of the WiFi functions will return a success flag (error code). In this simple program we ignored the return values, assuming it was ok. In the version on

the web, the process is restarted on error.

```
#define SSID_NAME "Valvano" // AP to connect to      1
#define ATYPE 'a' // analog data type                2
#define IP_ADDR 0xC0A80065 // server IP              3
#define PORT_NUM 5001 // Port number to be used      4
#define BUF_SIZE 12 //                               5
UINT8 uBuf[BUF_SIZE]; // UDP packet payload         6
int main(void){
    UINT8 IsDHCP = 0;
    _NetCfgIPv4Args_t ipV4;
    SockAddrIn_t Addr;
    UINT16 AddrSize = 0;
    INT16 SockID = 0;
    UINT32 data;
    unsigned char len = sizeof(_NetCfgIPv4Args_t);
    initClk(); // PLL 50 MHz, ADC needs PPL active    15
    ADC0_InitSWTriggerSeq3(7); // Ain7 is on PD0     16
    sl_Start(0, 0, 0); // Initializing the CC3100 device 17
    WlanConnect(); // connect to AP                   18
    sl_NetCfgGet(SL_IPV4_STA_P2P_CL_GET_INFO,&IsDHCP,&len, // 19
                (unsigned char *)&ipV4); // 20
    Addr.sin_family = SL_AF_INET; // 21
    Addr.sin_port = sl_Htons((UINT16)PORT_NUM); // 22
    Addr.sin_addr.s_addr = sl_Htonl((UINT32)IP_ADDR); // 23
    AddrSize = sizeof(SockAddrIn_t); // 24
    SockID = sl_Socket(SL_AF_INET,SL_SOCKET_DGRAM, 0); // 25
    while(1){
        uBuf[0] = ATYPE; // analog data type          26
        uBuf[1] = '='; //                               27
        data = ADC0_InSeq3(); // 0 to 4095, Ain7 is on PD0 28
        Int2Str(data,(char*)&uBuf[2]); // 6 digit number 29
        sl_SendTo(SockID, uBuf, BUF_SIZE, 0, // 30
                 (SockAddr_t *)&Addr, AddrSize); // 31
        ROM_SysCtlDelay(ROM_SysCtlClockGet() / 25); // 40ms 32
    }
}
```

*Program 9.3. Client software that measures ADC data and sends UDP packets.*

Program 9.4 shows the server software, which accepts UDP packets and plots the data on an ST7735 graphics LCD. Line 1 specifies the name of the access point (AP) to which the node will connect. The client and server use the same AP, which I

named **Valvano**, so I used the manual method to define the connection between the node and AP. The UDP payload will have a type field, which is defined in line 2. Lines 16, 22-25 configure the WiFi connection in a similar way as the client. Lines 17-20 initialize the ST7735 LCD and output a welcome message. Line 21 configures the LCD graphics routines specifying the range on the y-axis of the plot. Raw ADC data will be plotted versus time. Lines 26-29 define an IP address and port to use. Line 31 defines and opens a socket, and lines 32-33 bind the port to that socket. Lines 34-35 receive a UDP packet. Just like the client, we leave the socket open. If we wished to save power, we could close the socket, go into low-power mode, and reopen the connection after sleeping. Lines 36-51 decode the packet and plot the data on the LCD.

```

#define SSID_NAME "Valvano" // AP to connect to           1
#define ATYPE 'a' // analog data type                   2
#define IP_ADDR 0xC0A80065 // server IP                 3
#define PORT_NUM 5001 // Port number to be used         4
#define BUF_SIZE 12 //                                   5
UINT8 uBuf[BUF_SIZE]; // UDP packet payload            6
int main(void){
  UINT8 IsDHCP = 0;
  _NetCfgIPv4Args_t ipV4;
  SISockAddrIn_t Addr, LocalAddr;
  UINT16 AddrSize = 0;
  INT16 SockID = 0;
  INT16 Status = 1; // ok
  UINT32 data;
  unsigned char len = sizeof(_NetCfgIPv4Args_t);
  initClk(); // PLL 50 MHz, ADC needs PPL active         16
  ST7735_InitR(INTR_REDTAB); // Initialize              17
  ST7735_OutString("Internet of Things\n"); //          18
  ST7735_OutString("Embedded Systems\n"); //           19
  ST7735_OutString("Vol. 2, Valvano"); //              20
  ST7735_PlotClear(0,4095); // range from 0 to 4095    21
  sl_Start(0, 0, 0); // Initializing the CC3100 device  22
  WlanConnect(); // connect to AP                      23
  sl_NetCfgGet(SL_IPV4_STA_P2P_CL_GET_INFO,&IsDHCP,&len, // 24
    (unsigned char *)&ipV4); //                       25
  LocalAddr.sin_family = SL_AF_INET; //                 26
  LocalAddr.sin_port = sl_Htons((UINT16)PORT_NUM); //  27
  LocalAddr.sin_addr.s_addr = 0; //                   28
  AddrSize = sizeof(SISockAddrIn_t); //               29
  while(1){
    SockID = sl_Socket(SL_AF_INET,SL_SOCKET_DGRAM, 0); // 31
    Status = sl_Bind(SockID, (SISockAddr_t *)&LocalAddr, // 32
      AddrSize); //                                   33

```

```

Status = sl_RecvFrom(SocketID, uBuf, BUF_SIZE, 0, // 34
    (SISockAddr_t *)&Addr, (SISocklen_t *)&AddrSize );// 35
if((uBuf[0]==ATYPE)&&(uBuf[1]==' ')){ // 36
    int i,bOk; uint32_t place; // 37
    data = 0; bOk = 1; // 38
    i=4; // ignore possible negative sign 39
    for(place = 1000; place; place = place/10){ // 40
        if((uBuf[i]&0xF0)==0x30){ // ignore spaces 41
            data += place*(uBuf[i]-0x30); // 42
        }else{ // 43
            if((uBuf[i]&0xF0)!= ' '){ // 44
                bOk = 0; // 45
            } // 46
        } // 47
        i++; // 48
    } // 49
    if(bOk){ // 50
        ST7735_PlotLine(data); // 51
        ST7735_PlotNextErase(); // 51
    }
}
}
}
}
}

```

*Program 9.4. Server software that receives UDP packets and plots results on the LCD.*

Since UDP transmission is “best effort” we could lose packets or receive packets out of order. In this simple example we will not know if either of these errors were to occur. If we wished to have a more reliable transmission, we could have used TCP. Program 9.4 line 31 would have specified a socket stream instead of a datagram. To create a TCP communication, use the example software in the `tcp_socket` folder.

```

SocketID = sl_Socket(SL_AF_INET,SL_SOCKET_STREAM, 0); // TCP socket

```

## 9.4.5. Other CC3100 Applications

This section lists the sample applications are also provided for MSP430F5739, TM4C123GH6PM and SimpleLink Studio. The source code for these examples can be found in the examples directory after downloading CC3100SDK, the SimpleLink Wi-Fi CC3100 Software Development Kit (SDK) from the TI website. For more details on each example, see the docs folder included in the CC3100SDK download. The CC3100 comes preloaded with CC3100 BoosterPack comes preloaded with Out of Box HTML pages. Out of box demo highlights the following features: Simple WLAN Connection Using Smart Config, and easy access to CC3100 using mDNS and

HTTP Server.

**Antenna Selection.** This is a reference implementation for antenna-selection scheme running on the host MCU, to enable improved radio performance inside buildings

**Connection Policies.** This application demonstrates the usage of the CC3100 profiles and connection-policies.

**Send Email.** This application sends an email using SMTP to a user-configurable email address at the push of a button.

**Enterprise Network Connection.** This application demonstrates the procedure for connecting the CC3100 to an enterprise network.

**File Download.** This application demonstrates file download from a cloud server to the on board serial Flash.

**File System.** This application demonstrates the use of the file system API to read and write files from the serial Flash.

**Get Time.** This application connects to an SNTP cloud server and receives the accurate time.

**Get Weather.** This application connects to ‘Open Weather Map’ cloud service and receives weather data.

**Getting Started in AP Mode.** This application configures the CC3100 in AP mode. It verifies the connection by pinging the connected client.

**Getting Started in Station Mode.** This application configures the CC3100 in STA mode. It verifies the connection by pinging the connected Access Point.

**HTTP Server.** This application demonstrates using the on-chip HTTP Server APIs to enable static and dynamic web page content.

**IP Configuration.** This application demonstrates how to enable static IP configuration instead of using DHCP.

**mDNS.** This application registers the mDNS service for broadcasting and attempts to get the service by the name broadcasted by another device.

**Mode Configuration.** This application demonstrates switching between STA and AP modes.

**NWP Filters.** This application demonstrates the configuration of Rx-filtering to reduce the amount of traffic transferred to the host, and to achieve lower power consumption.

**NWP Power Policy.** This application shows how to enable different power policies to reduce power consumption based on use case in the station mode.

**P2P (Wi-Fi Direct).** This application configures the device in P2P (Wi-Fi Direct) mode and demonstrates how to communicate with a remote peer device.

**Provisioning AP.** This application demonstrates the use of the on Chip HTTP server for Wi-Fi provisioning in AP Mode, building upon example application 7.8 above.

**Provisioning with SmartConfig.** This application demonstrates the usage of TI's SmartConfig™ Wi-Fi provisioning technology. The Wi-Fi Starter Application for iOS and Android is required to use this application. It can be downloaded from following link: <http://www.ti.com/tool/wifistarter> or from the Apple App store and Google Play.

**Provisioning with WPS.** This application demonstrates the usage of WPS Wi-Fi provisioning with CC3100.

**Scan Policy.** The application demonstrates the scan-policy settings in CC3100.

**SPI Diagnostics Tool.** This is a diagnostics application for troubleshooting the host SPI configuration.

**SSL/TLS.** The application demonstrates the usage of certificates with SSL/TLS for application traffic privacy and device or user authentication

**TCP Socket.** The application demonstrates simple connection with TCP traffic.

**Transceiver Mode.** The application demonstrates the CC3100 transceiver mode of operation.

**UDP Socket.** The application demonstrates simple connection with UDP traffic.

**XMPP Client.** The application demonstrates instant messaging using a cloud based XMPP server.

These were the steps I used to create the UDP communication example. I began with the starter application, `ek-tm4c123gxl-boost-cc3100_starter`. I first changed `SSID_NAME` to match our access point

```
#define SSID_NAME "Valvano" // AP to connect to
```

Next, I compiled, downloaded and ran this application onto two LaunchPad+CC3100 systems, observing the operating on PuTTY. The interpreter output should show it has connected and shows the IP assigned to these two nodes by the AP. I could run the **ping** command to check the WiFi connection to my AP.

Once I was sure my two LaunchPad+CC3100 systems could communicate with my AP, I made a copy of the starter application by copy-pasting the entire folder. I renamed this new folder to `ek-tm4c123gxl-boost-cc3100_basic_wifi_UDP`. I opened the new project in the compiler IDE and opened the **main.c** from the `udp_socket` example folder. I added and/or merged the source code from **main.c** of `udp_socket` into **starter.c** of the new project. The event handlers and the main project needed merging, but the **BsdUdpClient** and **BsdUdpServer** functions were simply added. I changed the IP address to match the address given to the server.

```
#define IP_ADDR 0xC0A80068
```

I then loaded a version that called the client (send UDP) on one system

```
while(1){ BsdUdpClient(PORT_NUM)};
```

and loaded a version that called the server (receive UDP) on the other system

```
while(1){ BsdUdpServer(PORT_NUM)};
```

I ran the two systems in the debugger to see that packets were being sent. I did not use SmartConfig, because I knew the name of the AP. The last step was to modify the client and server so the client collects data and the server displays it.

---

## 9.4. Bluetooth Fundamentals

**Bluetooth** is wireless medium and a data protocol that connects devices together over a short distance. Examples of Bluetooth connectivity include headset to phone, speaker to computer, and fitness device to phone/computer. Bluetooth is an important component of billions of products on the market today. Bluetooth operates from 1 to 100 meters, depending on the strength of the radio. Most Bluetooth devices operate up to a maximum of 10 meters. However, in order to improve battery life, many devices reduce the strength of the radio, and therefore save power by operating across distances shorter than 10 meters. If the computer or phone provides a bridge to the internet, a Bluetooth-connected device becomes part of the Internet of Things (IoT).

Bluetooth is classified as a **personal area network** (PAN) because it implements communication within the range of an individual person. Alternatively, devices within a Bluetooth network are usually owned or controlled by one person. When two devices on the network are connected, we often say the devices are **paired**.

At the highest level, we see Bluetooth devices implement profiles. A **profile** is a suite of functionalities that support a certain type of communication. For example, the **Advanced Audio Distribution Profile** (A2DP) can be used to stream data. The **Health Device Profile** (HDP) is a standard profile for medical devices. There are profiles for remote controls, images, printers, cordless telephones, health devices, hands free devices, and intercoms. The profile we will use in this chapter is the **generic attribute protocol** (GATT). Within the GATT there can be one or more services. Table 9.8 shows some of the services that have been developed.

Specification Name	Assigned Number
Alert Notification Service	0x1811
Automation IO	0x1815
Battery Service	0x180F
Blood Pressure	0x1810
Body Composition	0x181B
Bond Management	0x181E
Continuous Glucose Monitoring	0x181F
Current Time Service	0x1805
Cycling Power	0x1818
Cycling Speed and Cadence	0x1816
Device Information	0x180A



Environmental Sensing	0x181A
Generic Access	0x1800
Generic Attribute	0x1801
Glucose	0x1808
Health Thermometer	0x1809
Heart Rate	0x180D
HTTP Proxy	0x1823
Human Interface Device	0x1812
Immediate Alert	0x1802
Indoor Positioning	0x1821
Internet Protocol Support	0x1820
Link Loss	0x1803
Location and Navigation	0x1819
Next DST Change Service	0x1807
Object Transfer	0x1825
Phone Alert Status Service	0x180E
Pulse Oximeter	0x1822
Reference Time Update Service	0x1806
Running Speed and Cadence	0x1814
Scan Parameters	0x1813
Transport Discovery	0x1824
Tx Power	0x1804
User Data	0x181C
Weight Scale	0x181D

**Table 9.8. Adopted GATT services, <https://www.bluetooth.com/specifications/gatt/services>**

Within a service there may be one or more characteristics. A **characteristic** is user or application data that is transmitted from one device to another across the network. One of the attributes of a characteristic is whether it is **readable**, **writeable**, or both. We will use the **notify indication** to stream data from the embedded object to the smart phone. Characteristics have a **universally unique identifier** (UUID), which is a 128-bit (16-byte) number that is unique. BLE can use either 16-bit or 32-bit UUIDs. A specific UUID is used within the network to identify a specific characteristic.

Often a characteristic has one or more descriptors. **Descriptors** may be information like its name and its units. We will also see **handles**, which are a mechanism to identify characteristics within the device. A handle is a pointer to an internal data structure within the GATT that contains all the information about that characteristic. Handles are not passed across the Bluetooth network; rather, handles are used by the host and controller to keep track of characteristics. UUIDs are passed across the network. Figure 9.27 shows a GATT service with seven characteristics.

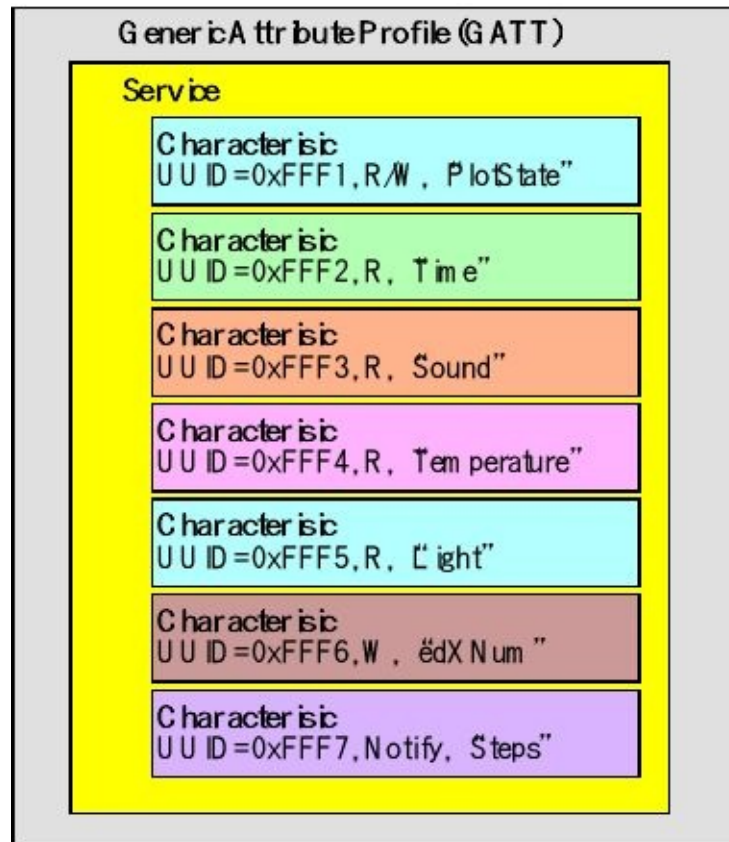
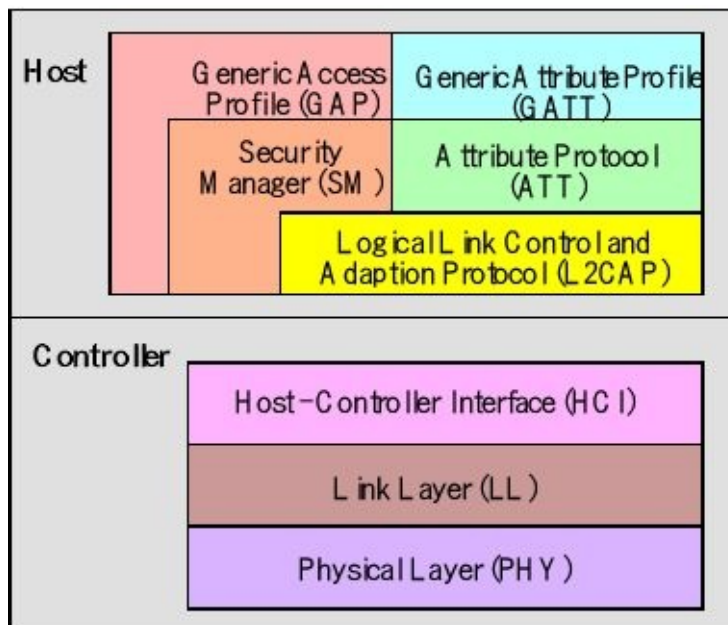


Figure 9.27. A GATT profile implements services, and a service has one or more characteristics.

### 9.4.1. Bluetooth Protocol Stack

The BLE protocol stack includes a controller and a host, as shown in Figure 9.28. Bluetooth BR (basic rate), Bluetooth EDR (enhanced data rate), and Bluetooth LE (low energy) all separate the controller and host as different layers and are often implemented separately. The user application and operating system sit on top of the host layer. This section is a brief overview of BLE. For more information on HCI, [www.ti.com/ble-wiki](http://www.ti.com/ble-wiki) and [www.ti.com/ble-stack](http://www.ti.com/ble-stack).



*Figure 9.28. The BLE stack. These layers are implemented inside the CC2650. The physical layer includes the antenna, which is outside the CC2650.*

The physical layer (PHY) is a 1Mbps adaptive frequency-hopping GFSK (Gaussian Frequency-Shift Keying) radio operating in the unlicensed 2.4 GHz ISM (Industrial, Scientific, and Medical) band.

The link layer (LL) controls the radiofrequency state of the device. The device can be in one of five states: standby, advertising, scanning, initiating, or connected. **Advertisers** transmit data without being in a connection, while scanners listen for advertisers. An **Initiator** is a device that is responding to an Advertiser with a connection request. If the Advertiser accepts, both the advertiser and initiator will enter a connected state. When a device is in a connection, it will be connected in one of two roles master or slave. The device that initiated the connection becomes the master, and the device that accepted the request becomes the slave. In Lab 6, the embedded system will be an advertiser and the smart phone will be the initiator.

The **host control interface** (HCI) layer provides a means of communication between the host and controller via a standardized interface. Standard HCI commands and events are specified in the Bluetooth Core Spec. The HCI layer is a thin layer which transports commands and events between the host and controller. In Lab 6, the HCI is implemented has function calls and callbacks within the CC2650 controller.

The **link logical control and adaption protocol** (L2CAP) layer provides data encapsulation services to the upper layers, allowing for logical end-to-end communication of data. The **security manager** (SM) layer defines the methods for pairing and key distribution, and provides functions for the other layers of the protocol stack to securely connect and exchange data with another device. The **generic access protocol** (GAP) layer handles the connection and security. In this simple example, we configure the GAP to setup and initiate advertisement. We will

use the GAP to connect our embedded system to a smart phone.

The overriding theme of Bluetooth communication is the exchange of data between paired devices. A service is a mechanism to exchange data. A collection of services is a profile. The **generic attribute profile** (GATT) handles services and profiles. The attribute protocol (ATT) layer protocol allows a device to expose “attributes” to other devices. All data communications that occur between two devices in a BLE connection are handled through the GATT.

The first step for our embedded device to perform is to configure and start advertisement, see Figure 9.29. In advertisement mode the device sends out periodic notifications of its existence and its willingness to connect. Another device, such as a smart phone, scans the area for possible devices. If desired this device can request a connection. If the advertiser accepts, both devices enter a connected phase, where the embedded device will be the slave (server) and the initiator becomes the master (client).

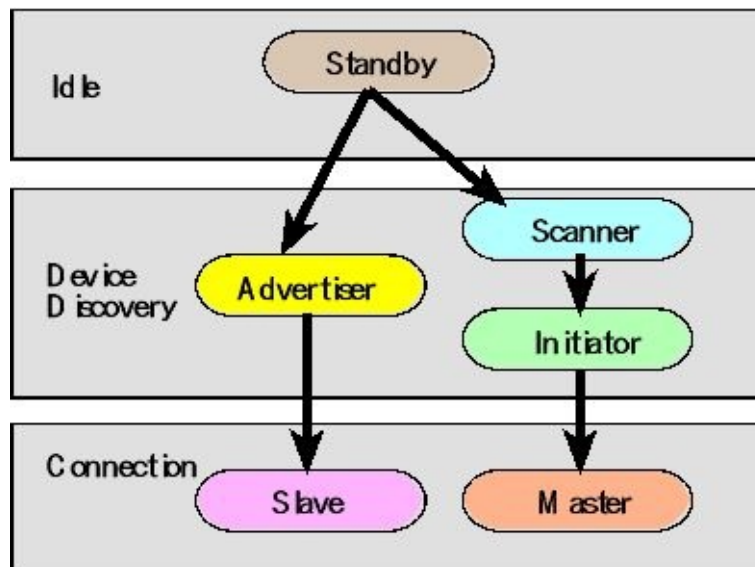


Figure 9.29. BLE connection steps.

In order to save power, the device spends most the time sleeping. The master sends out periodic requests to communicate. If the slave wishes to communicate, the master and slave will exchange data during this connection event. Figure 9.30 plots the device current versus time. This graph shows most of the current draw occurs during the connection events. The embedded device can save power by reducing the period of the connection events or by choosing not to participate in all the events.

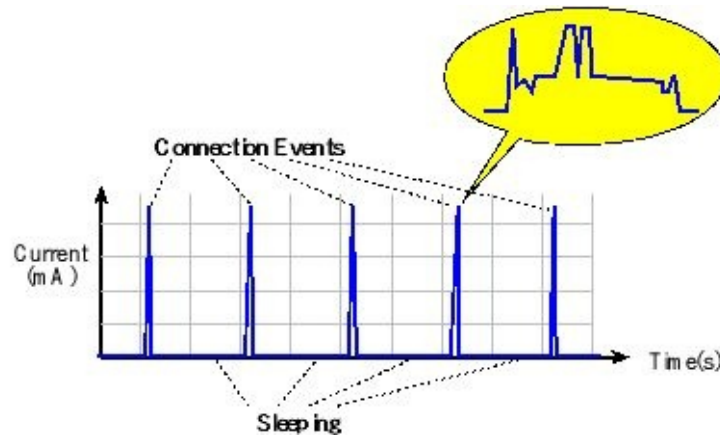


Figure 9.30. CC2650 current verses time, showing the connection events.

For example, you will see the advertising interval settings as parameters in the **NPI\_StartAdvertisement** message. In particular, the example projects set the advertising interval to 62.5ms.

## 9.4.2. Client-server Paradigm

The **client-server paradigm** is the dominant communication pattern for network protocols, see Figure 9.31. In general, the embedded system will be the server, and the smart phone will be the client. The client can request information from the server, or the client can send data to the server. With Bluetooth this exchange of data is managed by the services and profiles, discussed in the next section. There are four main profile types.

A **peripheral device** has sensors and actuators. On startup it advertises as connectable, and once connected it acts as a slave. In general, the embedded device will be a peripheral.

A **central device** has intelligence to manage the system. On startup it scans for advertisements and initiates connections. Once connected it acts as the master. In general, the smart phone will be a central device.

A **broadcaster** has sensors collecting information that is generally relevant. On startup it advertises but is not connectable. Other devices in the vicinity can read this information even though they cannot connect to the broadcaster. An example is a thermometer.

An **observer** can scan for advertisements but cannot initiate a connection. An example is a temperature display device that shows temperatures measured by broadcasters.

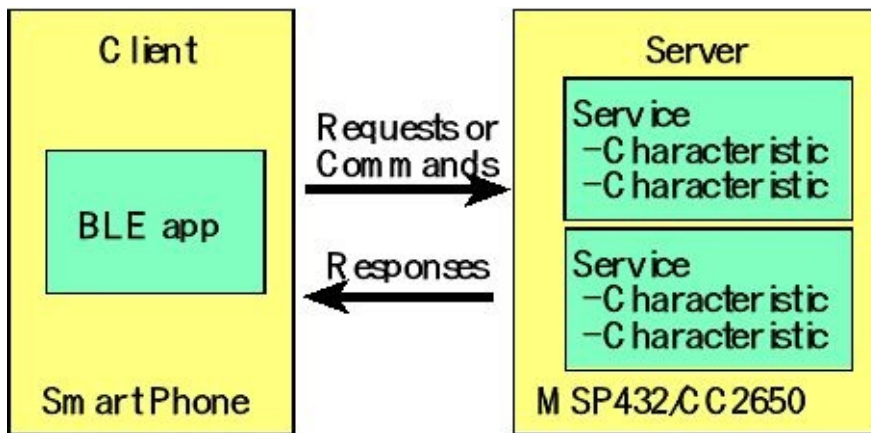


Figure 9.31. Client-server Paradigm.

**Read indication.** When the client wishes to know the value of a characteristic, it will issue a read indication. Inside the request will be a **universally unique identifier** (UUID) that specifies which characteristic is desired. The server will respond with the value by returning a **read confirmation**. The data may be one or more bytes. For large amounts of data, the response could be broken into multiple messages. In the example projects, the data will be 1, 2 or 4 bytes long. The size of the data is determined during initialization as the characteristic is configured.

**Write indication.** When the client wishes to set the value of a characteristic, it will issue a write indication. This request will include data. The request will also include a UUID that specifies to which characteristic the data should be written. The server will respond with an acknowledgement, called a **write confirmation**.

**Notify request.** When the client wishes to keep up to data on a certain value in the server, it will issue a notify request. The request includes a UUID. The server will respond with an acknowledgement, and then the server will stream data. This streaming could occur periodically, or it could occur whenever the value changes. In the example projects, **notify indication** messages are sent from server to client periodically. The client can start notification (listen command on the phone) or stop notifications.



## 9.5. CC2650 Solutions

### 9.5.1. CC2650 Microcontroller

There are three controllers on the CC2650: a main CPU, an RF core, and a sensor controller. Together, these combine to create a one-chip solution for Bluetooth applications. The **main CPU** includes 128kB of flash, 20kB of SRAM, and a full range of peripherals. Typically, the ARM Cortex-M3 processor handles the application layer and BLE protocol stack. However, in this chapter, we will place the application layer on another processor and use the CC2650 just to implement Bluetooth.

The **RF Core** contains an ARM Cortex-M0 processor that interfaces the analog RF and base-band circuitries, handles data to and from the system side, and assembles the information bits in a given packet structure. The RF core offers a high level, command-based API to the main CPU. The RF core is capable of autonomously handling the time-critical aspects of the radio protocols (802.15.4 RF4CE and ZigBee, Bluetooth Low Energy) thus offloading the main CPU and leaving more resources for the user application. The RF core has its own RAM and ROM. The ARM Cortex-M0 ROM is not programmable by customers. The basic circuit implementing the 2.4 GHz antenna is shown in Figure 9.32.

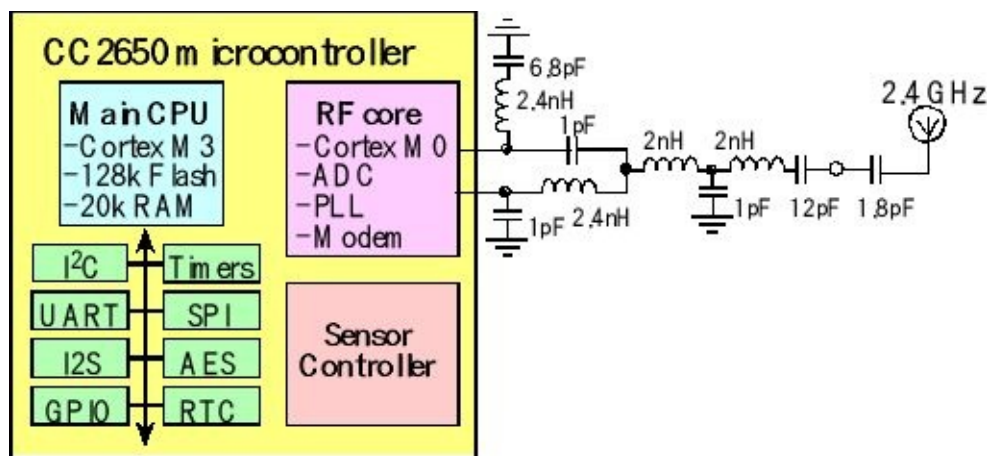


Figure 9.32. The CC2650 includes a main CPU, a suite of I/O devices, an RF core, and a sensor controller.

The **Sensor Controller** block provides additional flexibility by allowing autonomous data acquisition and control independent of the main CPU, further extending the low-power capabilities of the CC2650. The Sensor Controller is set up using a PC-based configuration tool, called Sensor Controller Studio, and example interfaces include:

- Analog sensors using integrated ADC

- Digital sensors using GPIOs, bit-banged I2C, and SPI
- UART communication for sensor reading or debugging
- Capacitive sensing
- Waveform generation
- Pulse counting
- Keyboard scan
- Quadrature decoder for polling rotation sensors
- Oscillator calibration

The CC2650 uses a radio-frequency (RF) link to implement Bluetooth Low Energy (BLE). As illustrated in Figure 9.33, the CC2650 can be used as a bridge between any microcontroller and Bluetooth. It is a transceiver, meaning data can flow across the link in both directions.

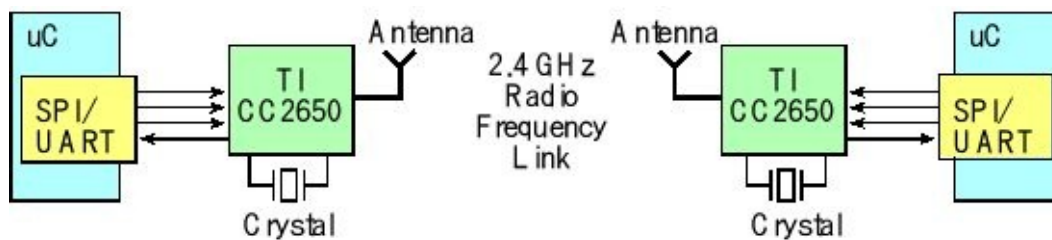


Figure 9.33. Block diagram of a wireless link between two microcontroller systems.

Figure 9.34 shows a CC2650 BoosterPack. This board comes preprogrammed with the simple network processor described in the next section. With a JTAG debugger, other programs can be loaded onto this CC2650. For more information, see

<http://www.ti.com/tool/boostxl-cc2650ma>



Figure 9.34. CC2650 BoosterPack (BOOSTXL-CC2650MA).

Figure 9.35 shows a CC2650 LaunchPad. The top part of the PCB is the debugger and the bottom part implements the CC2650 target system. To see the pin connections, see





Figure 9.35. CC2650 LaunchPad (LAUNCHXL-CC2650).

## 9.5.2. Single Chip Solution, CC2650 LaunchPad

The CC2650 microcontroller is a complete System-on-Chip (SoC) Bluetooth solution, as shown in Figure 9.36. One could deploy the application, the Bluetooth stack, and the RF radio onto the CC2650.

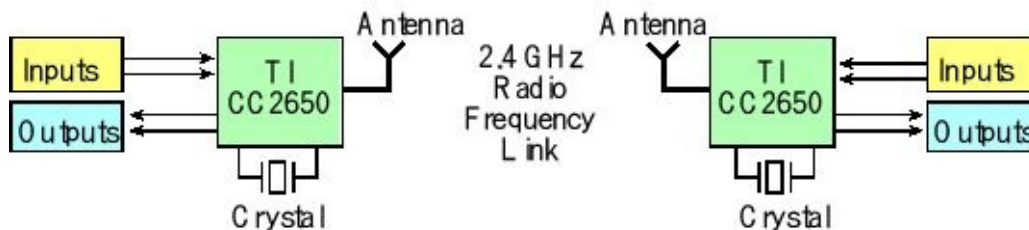


Figure 9.36. Block diagram of a wireless link between two single-chip embedded systems.

## 9.6. Network Processor Interface (NPI)

### 9.6.1. Overview

**Simple Network Processor (SNP)** is TI's name for the application that runs on the CC2650 when using the CC2650 with another microcontroller such as the MSP432 or TM4C123. In this configuration the controller and host are implemented together on the CC2650, while the profiles and application are implemented on an external MCU. The application and profiles communicate with the CC2650 via the **Application Programming Interface (API)** that simplifies the management of the BLE network processor. The SNP API communicates with the BLE device using the **Network Protocol Interface (NPI)** over a serial (SPI or UART) connection. In this chapter, we will use a UART interface as shown in Figure 9.37. This configuration is useful for applications that wish to add Bluetooth functionality to an existing device. In this paradigm, the application runs on the existing microcontroller, and BLE runs on the CC2650. For a description of the Simple Network Processor, refer to

SNP Developer guide [http://processors.wiki.ti.com/index.php/CC2640\\_BLE\\_Network\\_Processor](http://processors.wiki.ti.com/index.php/CC2640_BLE_Network_Processor)  
TI wiki page <http://www.ti.com/lit/ug/swru393c/swru393c.pdf>  
TI wiki page <http://processors.wiki.ti.com/index.php/NPI>

In this chapter, our TM4C123/MSP432 LaunchPad will be the application processor (AP) and the CC2650 will be the network processor (NP). There are 7 wires between the AP and the NP. Two wires are power and ground, one wire is a negative logic reset, two wires are handshake lines, and two wires are UART transmit and receive.

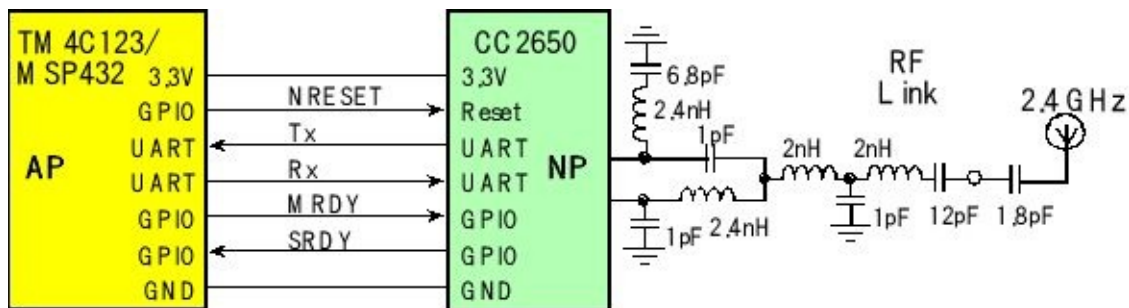


Figure 9.37. Hardware interface between the LaunchPad AP and the CC2650 NP.

To initialize Bluetooth, the master (AP) first resets the slave (NP). The **reset** line is a GPIO output of the AP and is the hardware reset line on the NP. There are two handshake lines: master ready and slave ready. **Master ready (MRDY)** is a GPIO output of the AP and a GPIO input to the NP. **Slave ready (SRDY)** is a GPIO output of the NP and a GPIO input of the AP. If the AP wishes to reset the NP, it sets MRDY

high and pulses reset low for 10 ms, Figure 9.38. Normally, the reset operation occurs once, and thereafter the reset line should remain high.

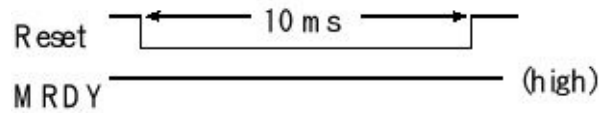


Figure 9.38. The LaunchPad AP can reset the CC2650 NP.

There are two types of communication. Messages can be sent from master to slave, or from slave to master. If the master (AP) wishes to send a message to the slave (NP), it follows 5 steps, Figure 9.39. First, the master sets MRDY low (Master: “I wish to send”). Second, the slave responds with SRDY low (Slave: “ok, I am ready”). The communication is **handshaked** because the master will wait for SRDY to go low. Third, the master will transmit a message on its UART output (Rx input to slave). The format of this message will be described later. Fourth, after the message has been sent, the master pulls MRDY high (Master: “I am done”). Fifth, the slave pulls its SRDY high (Slave: “ok”). Again, the handshaking requires the master to wait for SRDY to go high.

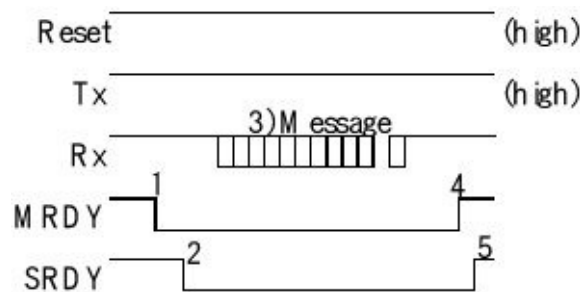


Figure 9.39. The LaunchPad AP can send a message to the CC2650 NP. Handshake means the steps 1 – 5 always occur in this sequence.

If the slave (NP) wishes to send a message to the master (AP), there are also 5 steps, Figure 9.40. First, the slave sets SRDY low (Slave: “I wish to send”). Second, the master responds with MRDY low (Master: “ok, I am ready”). You will notice in the example projects that the master will periodically check to see if the SRDY line has gone low, and if so it will receive a message. Third, the slave will transmit a message on its UART output (Tx output from slave). The format of this message will be the same for all messages. Fourth, after the message has been sent, the slave pulls SRDY high (Slave: “I am done”). The master will wait for SRDY to go high. Fifth, the master pulls its MRDY high (Master: “ok”).

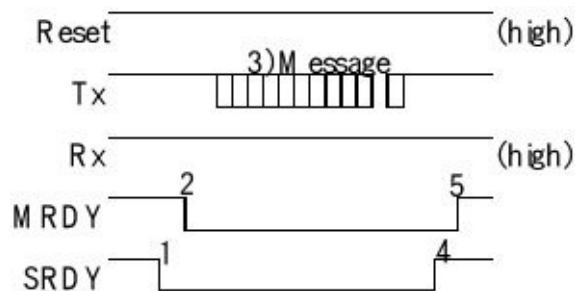


Figure 9.40. The CC2650 NP can send a message to the LaunchPad AP. Handshake means the steps 1 – 5 always occur in this sequence.

The format of the message is shown in Figure 9.41. The boxes in the figure represent UART frames. Each UART frame contains 1 start bit, 8 data bits, and 1 stop bit, sent at 115,200 bits/sec. All messages begin with a *start of frame* (SOF), which is a 254 (0xFE). The next two bytes are the payload length in little endian format. Since all the payloads in this chapter are less than 256 bytes, the second byte is the length, *L*, and the third byte is 0. The fourth and fifth bytes are the command. Most commands have a payload, which contains the parameters of the command. Some commands do not have a payload. All messages end with a **frame check sequence** (FCS). The FCS is the 8-bit exclusive or of all the data, not including the SOF and the FCS itself.

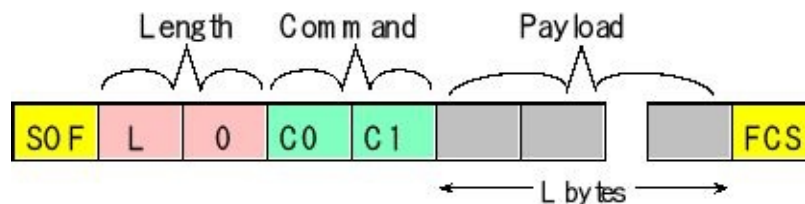


Figure 9.41. The format of an NPI message.

The following steps occur in this order

1. Initialize GATT (add services, characteristics, CCCD's);
2. Initialize GAP (advertisement data, connection parameters);
3. Advertise and optionally wait for a connection;
4. Respond to GATT requests and send notifications / indications as desired.

## 9.6.2. Services and Characteristics

After the CC2650 is reset, the next step is to services and characteristics. In the example projects we will define one service with multiple characteristics. To create a service, the master first issues an **Add Service** command (0x35,0x81). For each characteristic, the master sends an **Add Characteristic Value** (0x35,0x82) and an **Add Characteristic Description** (0x35,0x83) message. Once all the characteristics are defined, the master sends a **Register Service** command (0x35,0x84). Each of the commands has an acknowledgement response. The debugger output for a service with one characteristic is shown in Figure 9.42. The detailed syntax of these messages can be found in the TI CC2640 Bluetooth low energy Simple Network Processor API Guide.

### Add service

```
LP->SNP FE,03,00,35,81,01,F0,FF,B9
```

```
SNP->LP FE,01,00,75,81,00,F5
```

### Add CharValue 1

```
LP->SNP FE,08,00,35,82,03,0A,00,00,00,02,F1,FF,BA
```

```

SNP->LP FE,03,00,75,82,00,1E,00,EA
Add CharDescriptor1
LP->SNP FE,0B,00,35,83,80,01,05,00,05,00,44,61,74,61,00,0C
SNP->LP FE,04,00,75,83,00,80,1F,00,6D
Register service
LP->SNP FE,00,00,35,84,B1
SNP->LP FE,05,00,75,84,00,1C,00,29,00,C1

```

Figure 9.42. TExaSdisplay output as the device sets up a service with one characteristic. These data were collected running the *VerySimpleApplicationProcessor\_xxx* project.

Figures 9.43 through 9.46 show the four messages used to define a service with one characteristic. The **add service** creates a service. The **add characteristic value declaration** defines the read/write/notify properties of a characteristic in that service. The response to this message includes the handle. The **add characteristic description declaration** defines the name of the characteristic. When we create services with multiple characteristics, we simply repeat the “add characteristic value” and “add characteristic description” declarations for each. The **register service** makes that service active.

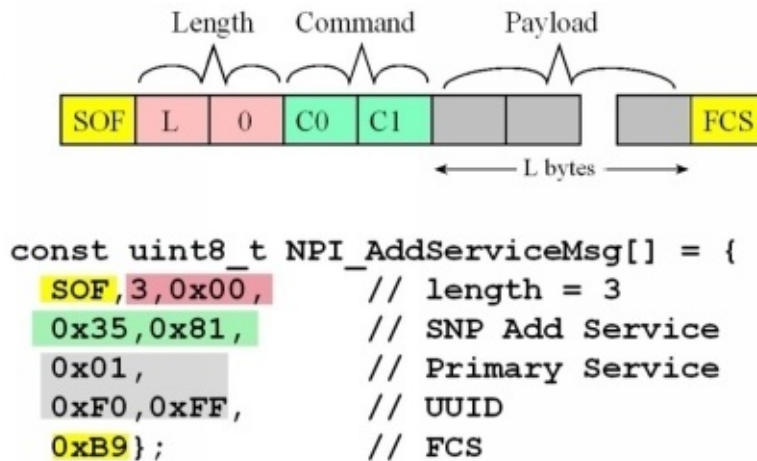


Figure 9.43. Add service message from the *VerySimpleApplicationProcessor\_xxx* project.

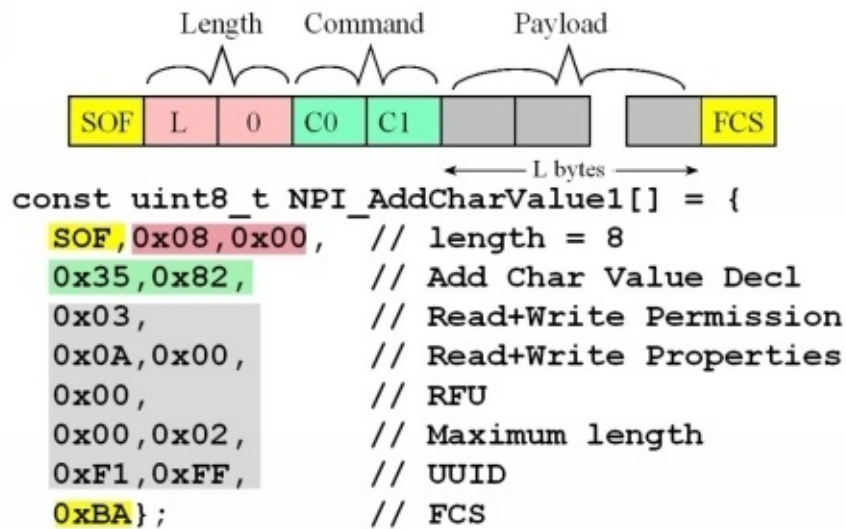


Figure 9.44. Add characteristic value declaration message from the *VerySimpleApplicationProcessor\_xxx* project.

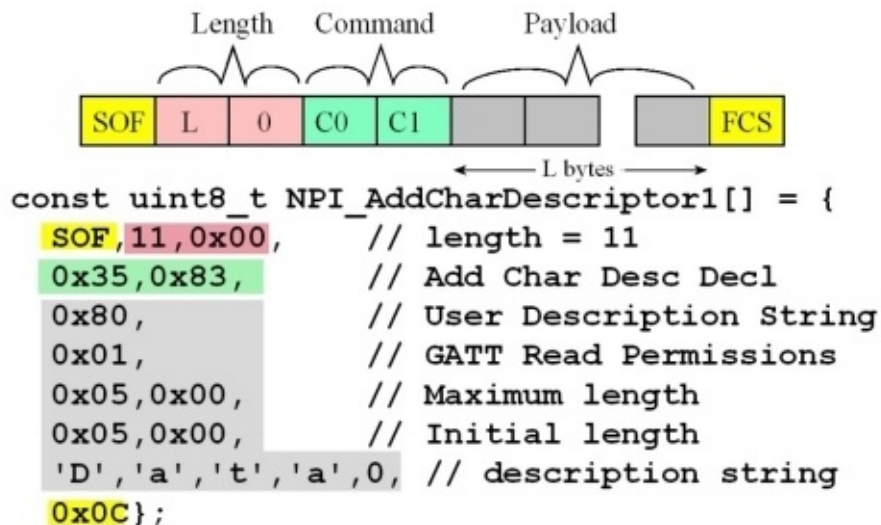


Figure 9.45. Add characteristic declaration message from the *VerySimpleApplicationProcessor\_xxx* project.

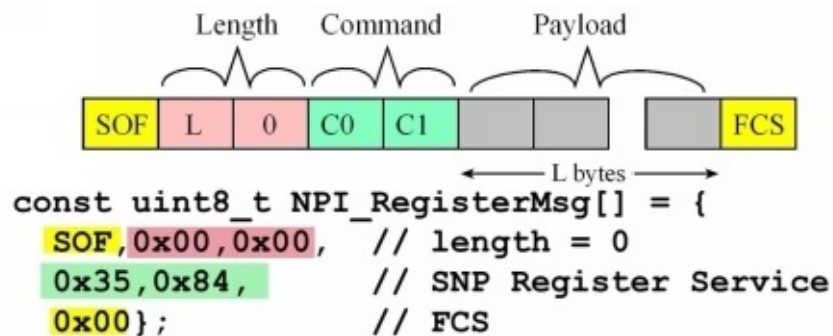


Figure 9.46. Register service message from the *VerySimpleApplicationProcessor\_xxx* project.



### 9.6.3. Advertising

After all the services and characteristics are defined, the master will setup and initiate advertising. The master will send four messages to set up advertising. The debugger output for advertising is shown in Figure 9.47. Each message will be acknowledged by the NP. A 0x35,0x85 message will set the device name. There are two 0x55,0x43 messages to configure the parameters of the advertising. The 0x55,0x42 message will start advertising. Again, detailed syntax of these messages can be found in the TI CC2640 Bluetooth low energy Simple Network Processor API Guide. Figure 9.48 shows the C code to define a **Set Device Name** message.

#### GATT Set DeviceName

LP->SNP FE,12,00,35,8C,01,00,00,53,68,61,70,65,20,74,68,65,20,57,6F,72,6C,64,DE

SNP->LP FE,01,00,75,8C,00,F8

#### SetAdvertisement1

LP->SNP FE,0B,00,55,43,01,02,01,06,06,FF,0D,00,03,00,00,EE

SNP->LP FE,01,00,55,43,00,17

#### SetAdvertisement2

LP->SNP FE,1B,00,55,43,00,10,09,53,68,61,70,65,20,74,68,65,20,57,6F,...,00,0C

SNP->LP FE,01,00,55,43,00,17

#### StartAdvertisement

LP->SNP FE,0E,00,55,42,00,00,00,64,00,00,00,01,00,00,00,C5,02,BB

SNP->LP FE,03,00,55,05,08,00,00,5B

Figure 9.47. TExaSdisplay output as the device sets up advertising. These data were collected running the *VerySimpleApplicationProcessor\_xxx* project.

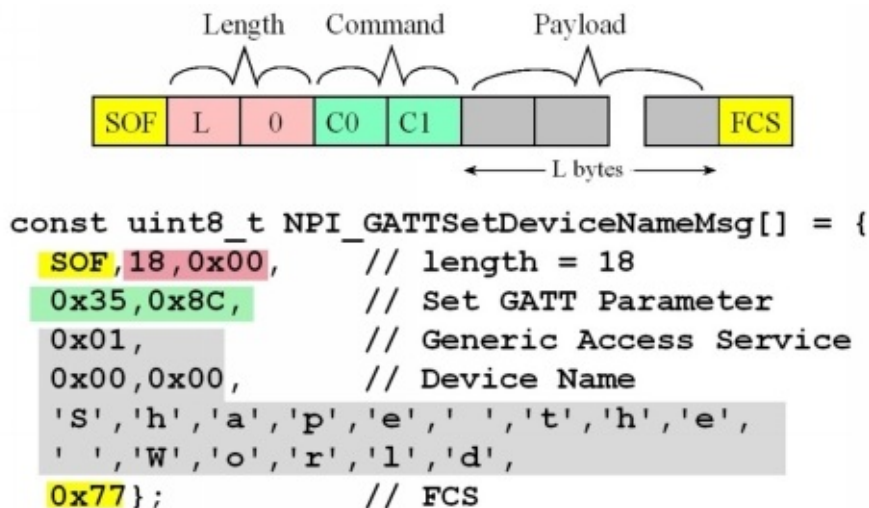


Figure 9.48. A set device name message from the *VerySimpleApplicationProcessor\_xxx* project.

### 9.6.4. Read and Write Indications

Figure 9.49 shows the message exchange when the client issues a read request. The

NP sends a **read indication** to the AP, containing the connection and handle of the characteristic. The AP responds with a read confirmation containing status, connection, handle, and the data.

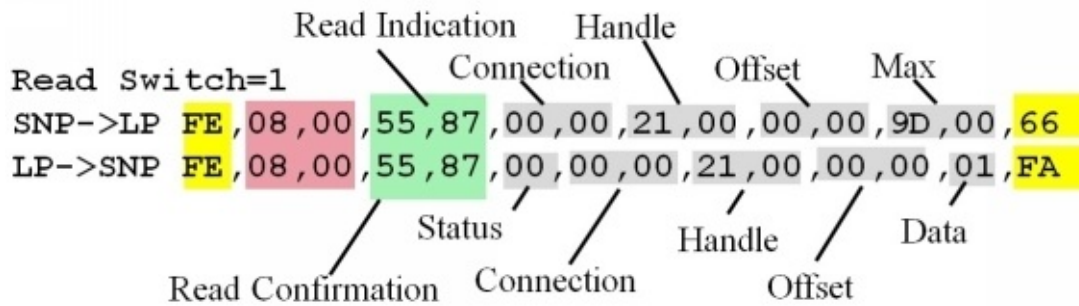


Figure 9.49. TExaSdisplay output occurring when the client issues a read request. These data were collected running the *VerySimpleApplicationProcessor\_xxx* project.

Figure 9.50 shows the message exchange when the client issues a write request. The NP sends a **write indication** to the AP, containing the connection, handle of the characteristic, and the data to be written. The AP responds with a **write confirmation** containing status, connection, and handle.

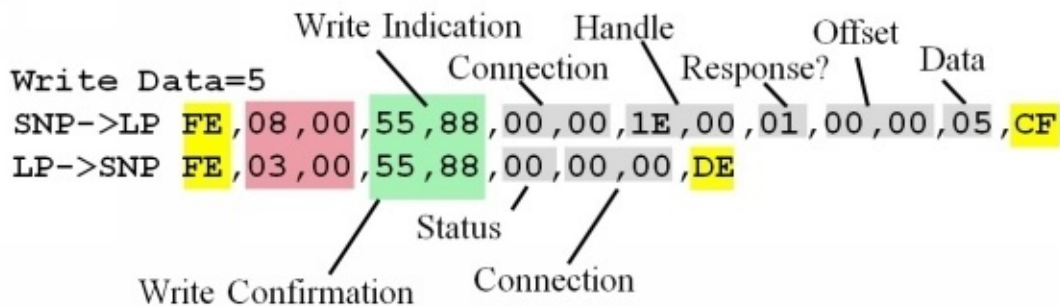


Figure 9.50. TExaSdisplay output occurring when the client issues a write request. These data were collected running the *VerySimpleApplicationProcessor\_xxx* project.



---

## 9.7. Application Layer Protocols for Embedded Systems

### 9.7.1. CoAP

The **Constrained Application Protocol** (CoAP) was specifically developed to allow resource-constrained devices to communicate over the Internet using UDP instead of TCP. In particular, many embedded devices have limited memory, processing power, and energy storage. Developers can interact with any CoAP-enabled device the same way they would with a device using a traditional Representational state transfer (REST) based API like HTTP. CoAP is particularly useful for communicating with low-power sensors and devices that need to be controlled via the Internet.

CoAP is a simple request/response protocol very similar to HTTP, that follows a traditional client/server model. Clients can make GET, PUT, POST, and DELETE requests to resources. CoAP packets use bitfields to maximize memory efficiency, and they make extensive usage of mappings from strings to integers to keep the data packets small enough to transport and interpret on-device. A CoAP message header is only 4-bytes long with most control messages being just that length. Most optional fields in the message format are in binary with the payload restricted in size so all CoAP messages fit inside a UDP datagram.

TCP is a connection oriented protocol, which means the server, or a client, will open a socket and establish a connection with the server. And the communication is done over a connection. For the duration of the communication, the connection is on. Whereas, COAP works on UDP, which means that it's connectionless. And it allows what we call as a disconnected operation, which means that the client and the server are not connected to each other. And therefore, they can act asynchronously.

Aside from the extremely small packet size, another major advantage of CoAP is its usage of UDP; using datagrams allows for CoAP to be run on top of packet-based technologies like SMS. There is a one-to-one mapping between CoAP and HTTP effectively providing a bridge between the all popular HTTP protocol to the emerging CoAP protocol.

All CoAP messages can be marked as either “confirmable” or “nonconfirmable,” serving as an application-level Quality of Service (QoS) to provide reliability. While SSL/TLS encryption isn't available over UDP, CoAP makes use of Datagram Transport Layer Security (DTLS), which is analogous to the TCP version of TLS. The default level of encryption is equivalent to a 3,072-bit RSA key. Even with all of this, CoAP is designed to work on microcontrollers with as little as 10KB of RAM.

One of the downsides of CoAP: It's a one-to-one protocol. Though extensions that

make group broadcasts possible are available, broadcast capabilities are not inherent to the protocol. Arguably, an even more important disadvantage is the need for both devices to be simultaneously powered, so when one sends a UDP, the other can receive it. In summary, the highlights of CoAP include:

- Small 4-byte header

- Option fields in binary

- Messages fit into one UDP datagram (no fragmentation)

- Works with SMS (text messaging)

- Connectionless

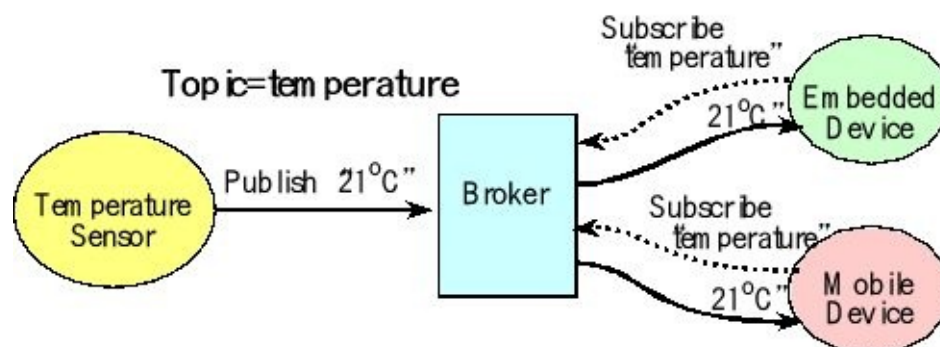
- Needs less than 10 kB of RAM

<http://www.infoworld.com/article/2972143/internet-of-things/real-time-protocols-for-iot-apps.html>

## 9.7.2 MQTT

**Message Queue Telemetry Transport (MQTT)** is a publish-subscribe messaging protocol, abbreviated as **pub-sub**. The MQTT name was inherited from a project at IBM. Similar to CoAP, it was built with resource-constrained devices in mind. MQTT has a lightweight packet structure designed to conserve both memory usage and power. A connected device subscribes to a topic hosted on an MQTT broker. Every time another device or service publishes data to a topic, all of the devices subscribed to it will automatically get the updated information.

Figure 9.51 shows the basic idea of the pub-sub model. MQTT uses an intermediary, which is called a **broker**. There are clients, or publishers, which produce data. The MQTT protocol calls this data a **topic**, and each topic must have a unique identifier. The figure shows a temperature sensor, which is an embedded device with a sensor attached, and it periodically publishes the topic “temperature”. To publish a topic means to send data to the broker. The broker keeps track of all the published information. Subscribers are devices consumers, which are interested in the data. What the subscribers do is they express their interest in a topic by sending a subscription message. In this figure we have two devices that have subscribed to the topic “temperature”. Whenever new data is available, the broker will serve it to both subscribers.



*Figure 9.51. With MQTT, the broker acts as an intermediary between producers and consumers.*

The fundamental advantage of a pub/sub model for communication in contrast with a client-server model is the decoupling of the communicating entities in space, time and synchronization. That is, the publisher and subscribed do not need to know each other, they do not run at the same time and they can act asynchronously. Other advantages of MQTT are the use of a publish-subscribe message queue and the many-to-many broadcast capabilities. Using a long-lived outgoing TCP connection to the MQTT broker, sending messages of limited bandwidth back and forth is simple and straightforward.

The downside of having an always-on connection is that it limits the amount of time the devices can be put to sleep. If the device mostly sleeps, then another MQTT protocol can be used: MQTT-SN, which is an extension of MQTT for sensor networks, originally designed to support ZigBee. MQTT-S is another extension that allows the use of UDP instead of TCP as the transport protocol, with support for peer-to-peer messaging and multicasting.

Another disadvantage of MQTT is the lack of encryption in the base protocol. MQTT was designed to be a lightweight protocol, and incorporating encryption would add a significant amount of overhead to the connection. One can however, use Transport Layer Security(TLS) extensions to TCP, or add custom security at the application level.

**References:**

<http://www.hivemq.com/blog/mqtt-essentials/>

<http://www.infoworld.com/article/2972143/internet-of-things/real-time-protocols-for-iot-apps.html>

---

## 9.8. Exercises

**9.1** Consider a wired communication system (like UART or CAN).

- a) Assume the signal has a rise time of 25 us. What is the approximate highest frequency component of this signal?
- b) Assuming a VF of 0.7, what is the wavelength of this highest frequency?
- c) Over what cable length would you have to consider this system as a transmission line?

**9.2** Consider a communication with a channel bandwidth of 10 kHz and an SNR of 60 dB. What is the maximum possible data transfer rate in bits/sec?

**9.3** What are there so many frequency bands for Bluetooth and WiFi?

**9.4** Consider bit-stuffing

- a) Define bit-stuffing
- b) Why do Ethernet and CAN implement bit-stuffing?
- c) UART does not implement bit-stuffing. How does the lack of bit-stuffing limit the UART?
- d) SPI does not implement bit-stuffing. Why does the lack of bit-stuffing not limit the SPI transmission in the same way as UART is limited?

**9.5** Consider how the ACK bit is used in a CAN network.

- a) What do the receivers do during the ACK bit?
- b) What does it mean if the ACK bit is dominant?
- c) What does it mean if the ACK bit is recessive?

**9.6** If the CAN channel is noisy, it is possible that some bits will be transmitted in error. Assume there are four nodes, one is transmitting and three are receiving. What happens if a data bit is flipped in the channel due to noise being added into the channel?

**9.7** Consider a situation where two microcontrollers are connected with a CAN network. Computer 1 generates 8-bit data packets that must be sent to computer 2, and computer 2 generates 8-bit data packets that must be sent to computer 1. The packets are generated at random times, and the goal is to minimize the latency between when a data packet is generated on one computer to when it is received on the other. Describe the CAN protocol you would use: 11-bit versus 29-bit ID, number of bytes of data, and bandwidth. Clearly describe what is in the ID and how the data is formatted.

**9.8** A CAN system has a baud rate of 100,000 bits/sec, 29-bit ID, and three bytes of

data per frame. Assuming there is no bit-stuffing, what is the maximum bandwidth of this network, in bytes/s.

**9.9** A CAN system has a baud rate of 200,000 bits/sec, 11-bit ID, and five bytes of data per frame. Assuming there is no bit-stuffing, what is the maximum bandwidth of this network, in bytes/s.

**9.10** Consider a situation where 4 microcontrollers are connected together using a CAN network. Assume for this question that each frame contains 100 bits. Also assume the baud rate is 100,000 bits/sec, therefore it takes 1ms to send a frame. Initially, the CAN controllers are initialized (i.e., all computers have previously executed **CAN\_Open**).

At time = 0                      computer A calls **CAN\_Send** with ID=1000

At time = 300  $\mu$ s                      computer B calls **CAN\_Send** with ID=800

At time = 500  $\mu$ s                      computer C calls **CAN\_Send** with ID=900

At time = 700  $\mu$ s                      computer D calls **CAN\_Send** with ID=600

Specify the time sequence in which the four frames occur on the CAN network. Clearly define the begin and end times when each message is visible on the CAN network.

**9.11** In a CAN network, what is the purpose of the **CRC** field? I.e., what is CRC used for?

**9.12** Why is BLE considered a personal area network, and WiFi is not?

**9.13** How does BLE achieve low energy?

**9.14** Define the following terms in 16 words or less as they apply to BLE.

- a) Service              b) Characteristic              c) Advertising
- d) Client              e) Server              f) Profile
- g) Stack              g) UUID              h) Handle
- i) Read indication              j) Write indication              k) Notify indication

# 10. Robotic Systems

## Chapter 10 objectives are to:

- Introduce the general approach to digital control systems
- Design and implement some simple closed-loop control systems
- Develop a methodology for designing PID control systems
- Present the terminology and give examples of fuzzy logic control system

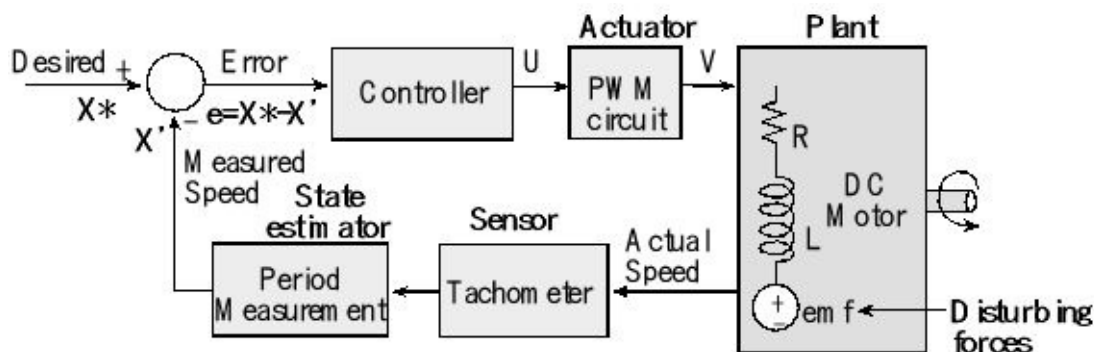
Throughout all three volumes of this series of books, we developed systems that collected information concerning the external environment. A typical application of embedded systems is to use this information in order to control the external environment. To build this microcontroller-based control system we will need an output device that the computer can use to manipulate the external environment. Control systems originally involved just analog electronic circuits and mechanical devices. With the advent of inexpensive yet powerful microcontrollers, implementing the control algorithm in software provided a lower cost and more powerful product. The goal of this chapter is to provide a brief introduction to this important application area. Control theory is a richly developed discipline, and most of this theory is beyond the scope of this book. Consequently, this chapter focuses more on implementing the control system with an embedded computer and less on the design of the control equations.

## 10.1. Introduction to Digital Control Systems

A **control system** is a collection of mechanical and electrical devices connected for the purpose of commanding, directing, or regulating a **physical plant** (see Figure 10.1). The **real state variables** are the properties of the physical plant that are to be controlled. The **sensor** and **state estimator** comprise a data acquisition system. The goal of this data acquisition system is to estimate the state variables. A **closed-loop** control system uses the output of the state estimator in a feedback loop to drive the errors to zero. The control system compares these **estimated state variables**,  $X'(t)$ , to the **desired state variables**,  $X^*(t)$ , in order to decide appropriate action,  $U(t)$ . The **actuator** is a transducer that converts the control system commands,  $U(t)$ , into driving forces,  $V(t)$ , that are applied to the physical plant. In general, the goal of the control system is to drive the real state variables to equal the desired state variables. In actuality though, the controller attempts to drive the estimated state variables to equal the desired state variables. It is important to have an accurate state estimator, because any differences between the estimated state variables and the real state variables will translate directly into controller errors. If we define the error as the difference between the desired and estimated state variables:

$$e(t) = X^*(t) - X'(t)$$

then the control system will attempt to drive  $e(t)$  to zero. In general control theory,  $X(t)$ ,  $X'(t)$ ,  $X^*(t)$ ,  $U(t)$ ,  $V(t)$  and  $e(t)$  refer to vectors, but the examples in this chapter control only a single parameter. Even though this chapter shows one-dimensional systems, and it should be straight-forward to apply standard multivariate control theory to more complex problems. We usually evaluate the effectiveness of a control system by determining three properties: steady state controller error, transient response, and stability. The **steady state controller error** is the average value of  $e(t)$ . The **transient response** is how long does the system take to reach 99% of the final output after  $X^*$  is changed. A system is **stable** if steady state (smooth constant output) is achieved. The error is small and bounded on a stable system. An **unstable** system oscillates, or it may saturate.



*Figure 10.1. Block diagram of a microcomputer-based closed-loop control system.*

An **open-loop** control system does not include a state estimator. It is called open loop because there is no feedback path providing information about the state variable to the controller. It will be difficult to use open-loop with the plant that is complex because the disturbing forces will have a significant effect on controller error. On the other hand, if the plant is well-defined and the disturbing forces have little effect, then an open-loop approach may be feasible. Because an open-loop control system does not know the current values of the state variables, large errors can occur. Stepper motors are often used in open loop fashion.



---

## 10.2. Binary Actuators

### 10.2.1. Electrical Interface

Relays, solenoids, and DC motors are grouped together because their electrical interfaces are similar. We can add speakers to this group if the sound is generated with a square wave. In each case, there is a coil, and the computer must drive (or not drive) current through the coil. To interface a coil, we consider **voltage**, **current** and **inductance**. We need a power supply at the desired voltage requirement of the coil. If the only available power supply is larger than the desired coil voltage, we use a voltage regulator (rather than a resistor divider to create the desired voltage.) We connect the power supply to the positive terminal of the coil, shown as +V in Figure 10.2. We will use a transistor device to drive the negative side of the coil to ground. The computer can turn the current on and off using this transistor. The second consideration is current. In particular, we must however select the power supply and an interface device that can support the coil current. The 7406 is an open collector driver capable of sinking up to 40 mA. The 2N2222 is a **bipolar junction transistor** (BJT), NPN type, with moderate current gain. The TIP120 is a **Darlington transistor**, also NPN type, which can handle larger currents. The IRF540 is a **MOSFET** transistor that can handle even more current. BJT and Darlington transistors are current-controlled (meaning the output is a function of the input current), while the MOSFET is voltage-controlled (output is a function of input voltage). When interfacing a coil to the microcontroller, we use information like Table 10.1 to select an interface device capable the current necessary to activate the coil. It is a good design practice to select a driver with a maximum current at least twice the required coil current. When the digital **Port** output is high, the interface transistor is active and current flows through the coil. When the digital **Port** output is low, the transistor is not active and no current flows through the coil.

Device	Type	Maximum current
TM4C	CMOS	8 mA (set bits in DR8R)
MSP432	CMOS	20 mA (DS=1, P2.0 – P2.3)
7406	TTL logic	40 mA
PN2222	BJT NPN	150 mA
2N2222	BJT NPN	500 mA
TIP120	Darlington NPN	5 A
IRF540	power MOSFET	28 A

**Table 10.1. Four possible devices that can be used to interface a coil to the microcontroller.**

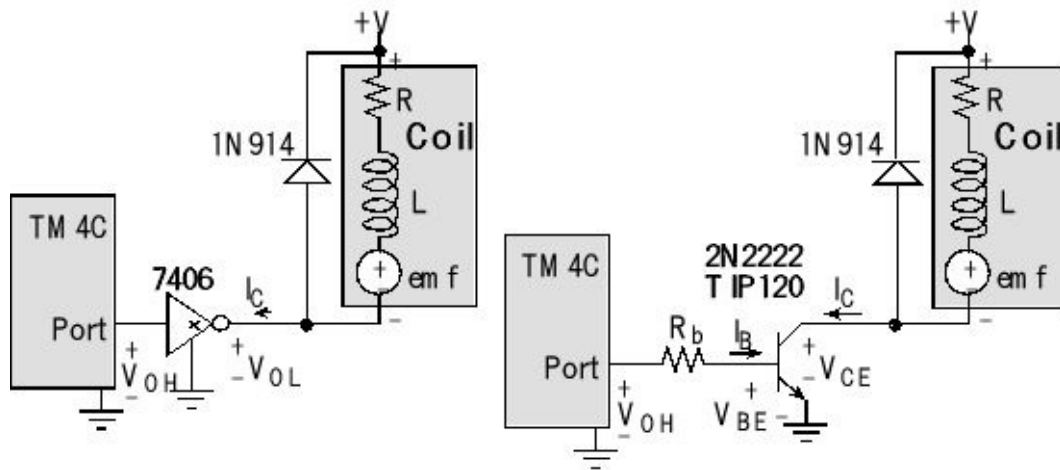


Figure 10.2. Binary interface to EM relay, solenoid, DC motor or speaker.

The third consideration is inductance in the coil. The 1N914 diode in Figure 10.1 provides protection from the **back emf** generated when the switch is turned off, and the large  $di/dt$  across the inductor induces a large voltage (on the negative terminal of the coil), according to  $V=L \cdot di/dt$ . For example, if you are driving 0.1A through a 0.1 mH coil (**Port** output = 1) using a 2N2222, then disable the driver (**Port** output = 0), the 2N2222 will turn off in about 20ns. This creates a  $di/dt$  of at least  $5 \cdot 10^6$  A/s, producing a back emf of 500 V! The 1N914 diode shorts out this voltage, protecting the electronics from potential damage. The 1N914 is called a **snubber diode**.

**Observation:** It is important to realize that many devices cannot be connected directly up to the microcontroller. In the specific case of motors, we need an interface that can handle the voltage and current required by the motor.

If you are sinking 16 mA ( $I_{OL}$ ) with the 7406, the output voltage ( $V_{OL}$ ) will be 0.4V. However, when the  $I_{OL}$  of the 7406 equals 40 mA, its  $V_{OL}$  will be 0.7V. 40 mA is not a lot of current when it comes to typical coils. However, the 7406 interface is appropriate to control small relays.

**Checkpoint 10.1:** A relay is interfaced with the 7406 circuit in Figure 10.2. The positive terminal of the coil is connected to +5V and the coil requires 40 mA. What will be the voltage across the coil when active?

When designing an interface, we need to know the desired coil voltage ( $V_{coil}$ ) and coil current ( $I_{coil}$ ). Let  $V_{be}$  be the base-emitter voltage that activates the NPN transistor and let  $h_{fe}$  be the current gain. There are three steps when interfacing an N-channel (right side of Figure 10.2.)

- 1) Choose the interface voltage  $V$  equal to  $V_{coil}$  (since  $V_{CE}$  is close to zero)
- 2) Calculate the desired base current  $I_b = I_{coil} / h_{fe}$  (since  $I_C$  equals  $I_{coil}$ )
- 3) Calculate the interface resistor  $R_b \leq (V_{OH} - V_{be}) / I_b$  (choose a resistor 2 to 5 times smaller)

With an N-channel switch, like Figure 10.2, current is turned on and off by connecting/disconnecting one side of the coil to ground, while the other side is fixed

at the voltage supply. A second type of binary interface uses P-channel switches to connect/disconnect one side of the coil to the voltage supply, while the other side fixed at ground, as shown in Figure 10.3. In other to activate a PNP transistor (e.g., PN2907 or TIP125), there must be a  $V_{EB}$  greater than 0.7 V. In order to deactivate a PNP transitory, the  $V_{EB}$  voltage must be 0. Because the transistor is a current amplifier, there must be a resistor into the base in order to limit the base current.

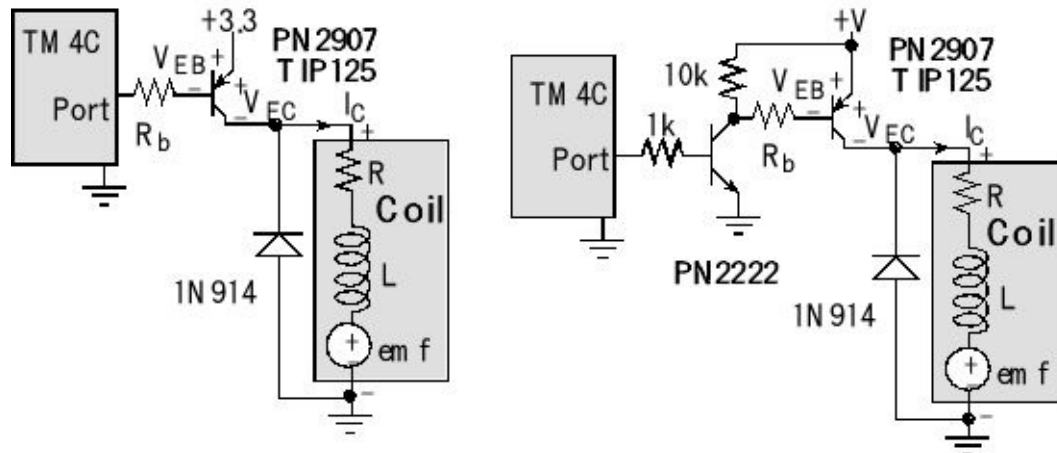


Figure 10.3. PNP interface to EM relay, solenoid, DC motor or speaker.

To understand how the PNP interface on the right of Figure 10.3 operates, consider the behavior for the two cases: the Port output is high and the Port output is low. If the Port output is high, its output voltage will be between 2.4 and 3.3 V. This will cause current to flow into the base of the PN2222, and its  $V_{be}$  will saturate to 0.7 V. The base current into the PN2222 could be from  $(2.4-0.7)/1000$  to  $(3.3-0.7)/1000$ , or 1.7 to 2.6 mA. The microcontroller will be able to source this current. This will saturate the PN2222 and its  $V_{CE}$  will be 0.3 V. This will cause current to flow out of the base of the PN2907, and its  $V_{EB}$  will saturate to 0.7 V. If the supply voltage is  $V$ , then the PN2907 base current is  $(V-0.7-0.3)/R_b$ . Since the PNP transistor is on,  $V_{EC}$  will be small and current will flow from the supply to the coil. If the port output is low, the voltage output will be between 0 and 0.4V. This not high enough to activate the PN2222, so the NPN transistor will be off. Since there is no  $I_C$  current in the PN2222, the 10k and  $R_b$  resistors will place  $+V$  at the base of the PN2907. Since the  $V_{EB}$  of the PN2907 is 0, this transistor will be off, and no current will flow into the coil.

MOSFETs can handle significantly more current than BJT or Darlington transistors. MOSFETs are voltage controlled switches. The difficulty with interfacing MOSFETs to a microcontroller is the large gate voltage needed to activate it. The left side of Figure 10.4 is an N-channel interface. The IRF540 N-channel MOSFET can sink up to 28A when the gate-source voltage is above 7V. This circuit is negative logic. When the port pin is high, the 2N2222 is active making the MOSFET gate voltage 0.3V ( $V_{CE}$  of the PN2222). A  $V_{GS}$  of 0.3V turns off the MOSFET. When the port pin is low, the 2N2222 is off making the MOSFET gate voltage  $+V$  (pulled up through the

10kΩ resistor). The  $V_{GS}$  is +V, which turns the MOSFET on.

The right side of Figure 10.4 shows a P-channel MOSFET interface. The IRF9540 P-channel MOSFET can source up to 20A when the source-gate voltage is above 7V. The FQP27P06 P-channel MOSFET can source up to 27A when the source-gate voltage is above 6V. This circuit is positive logic. When the port pin is high, the 2N2222 is active making the MOSFET gate voltage 0.3V. This makes  $V_{SG}$  equal to +V-0.3, which turns on the MOSFET. When the port pin is low, the 2N2222 is off. Since the 2N2222 is off, the 10kΩ pull-up resistor makes the MOSFET gate voltage +V. In this case  $V_{SG}$  equals 0, which turns off the MOSFET.

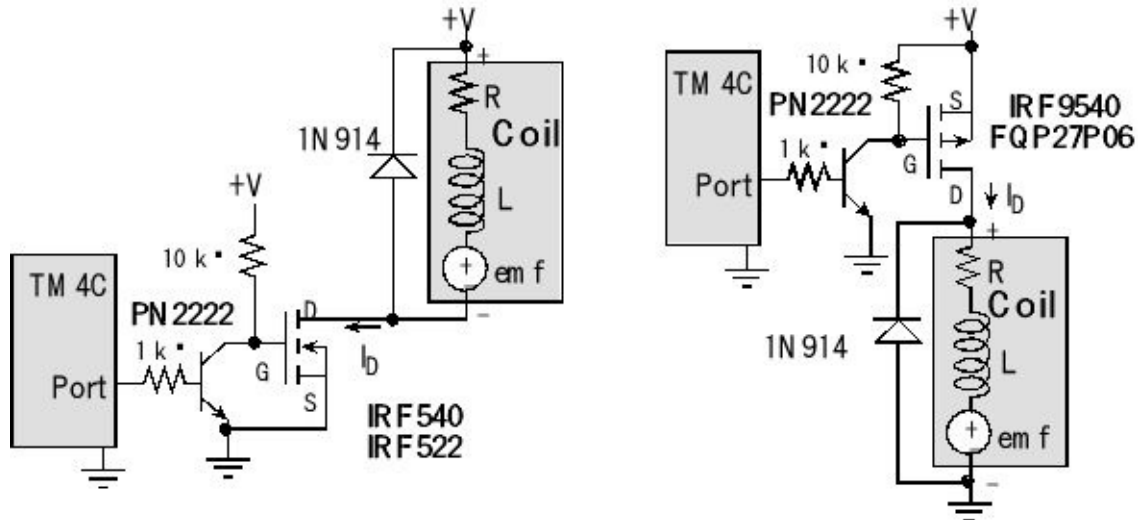


Figure 10.4. MOSFET interfaces to EM relay, solenoid, DC motor or speaker.

An H-bridge combines P-channel and N-channel devices allowing current to flow in either direction. Figures 4.26 and 4.27 in Volume 2 show applications of the L293 H-bridge, while Figure 10.5 shows one of the H-bridge circuits internal to the L293. If 1A is high,  $Q_1$  is on and  $Q_2$  is off. If 1A is low,  $Q_1$  is off and  $Q_2$  is on. 2A controls  $Q_3$  and  $Q_4$  in a similar fashion. If 1A is high and 2A is low, then  $Q_1$   $Q_4$  are on and current flows left to right across coil A. If 1A is low and 2A is high, then  $Q_2$   $Q_3$  are on and current flows right to left across coil A.

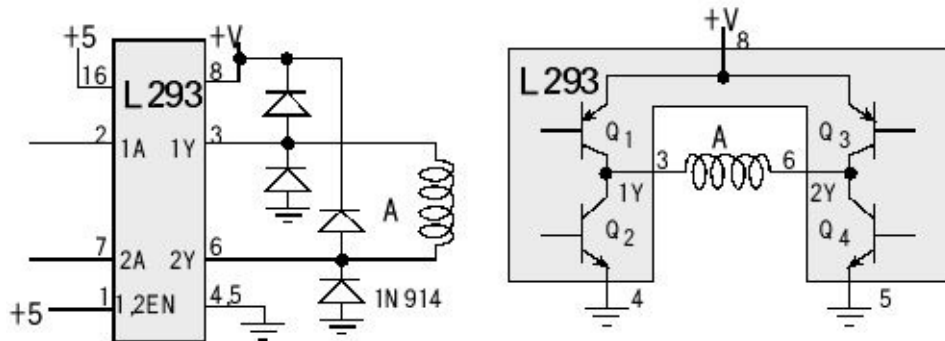


Figure 10.5. An H-bridge can drive current in either direction (the actual L293 uses all N-channel devices).

## 10.2.2. DC Motor Interface with PWM

Similar to the solenoid and EM relay, the DC motor has a frame that remains motionless, and an armature that moves. In this case, the armature moves in a circular manner (shaft rotation).

In the previous interfaces the microcontroller was able to control electrical power to a device in a binary fashion: either all on or all off. Sometimes it is desirable for the microcontroller to be able to vary the delivered power in a variable manner. One effective way to do this is to use pulse width modulation (PWM). The basic idea of PWM is to create a digital output wave of fixed frequency, but allow the microcontroller to vary its duty cycle. The system is designed in such a way that **High+Low** is constant (meaning the frequency is fixed). The **duty cycle** is defined as the fraction of time the signal is high:

$$\text{duty cycle} = \frac{\text{High}}{\text{High} + \text{Low}}$$

Hence, duty cycle varies from 0 to 1. We interface this digital output wave to an external actuator (like a DC motor), such that power is applied to the motor when the signal is high, and no power is applied when the signal is low. We purposely select a frequency high enough so the DC motor does not start/stop with each individual pulse, but rather responds to the overall average value of the wave. The average value of a PWM signal is linearly related to its duty cycle and is independent of its frequency. Let  $P$  ( $P=V*I$ ) be the power to the DC motor, Figures 10.2 - 10.5, when the PWM signal is high. Under conditions of constant speed and constant load, the delivered power to the motor is linearly related to duty cycle.

*Delivered Power =*

$$\text{duty cycle} * P = \frac{\text{High}}{\text{High} + \text{Low}} * P$$

Unfortunately, as speed and torque vary, the developed emf will affect delivered power. Nevertheless, PWM is a very effective mechanism, allowing the microcontroller to adjust delivered power.

A DC motor has an electro-magnet as well. When current flows through the coil, a magnetic force is created causing a rotation of the shaft. Brushes positioned between the frame and armature are used to alternate the current direction through the coil, so that a DC current generates a continuous rotation of the shaft. When the current is removed, the magnetic force stops, and the shaft is free to rotate. The resistance in the coil ( $R$ ) comes from the long wire that goes from the + terminal to the – terminal of the motor. The inductance in the coil ( $L$ ) arises from the fact that the wire is wound into coils to create the electromagnetics. The coil itself can generate its own voltage (emf) because of the interaction between the electric and magnetic fields. If the coil is a DC motor, then the emf is a function of both the speed of the motor and the developed torque (which in turn is a function of the applied load on the motor.)

Because of the internal emf of the coil, the current will depend on the mechanical load. For example, a DC motor running with no load might draw 50 mA, but under load (friction) the current may jump to 500 mA.

There are lots of motor driver chips, but they are fundamentally similar to the circuits shown in Figure 10.2. For the 2N2222 and TIP120 NPN transistors, if the port output is low, no current can flow into the base, so the transistor is off, and the collector current,  $I_C$ , will be zero. If the port output is high, current does flow into the base and  $V_{BE}$  goes above  $V_{BEsat}$  turning on the transistor. The transistor is in the linear range if  $V_{BE} \leq V_{BEsat}$  and  $I_c = h_{fe} \cdot I_b$ . The transistor is in the saturated mode if  $V_{BE} \geq V_{BEsat}$ ,  $V_{CE} = 0.3V$  and  $I_c < h_{fe} \cdot I_b$ . We select the resistor for the NPN transistor interfaces to operate right at the transition between linear and saturated mode. We start with the desired coil current,  $I_{coil}$  (the voltage across the coil will be  $+V - V_{CE}$  which will be about  $+V - 0.3V$ ). Next, we calculate the needed base current ( $I_b$ ) given the current gain of the NPN

$$I_b = I_{coil} / h_{fe}$$

knowing the current gain of the NPN ( $h_{fe}$ ), see Table 10.2. Finally, given the output high voltage of the microcontroller ( $V_{OH}$  is about 3.3 V) and base-emitter voltage of the NPN ( $V_{BEsat}$ ) needed to activate the transistor, we can calculate the desired interface resistor.

$$R_b \leq (V_{OH} - V_{BEsat}) / I_b = h_{fe} * (V_{OH} - V_{BEsat}) / I_{coil}$$

The inequality means we can choose a smaller resistor, creating a larger  $I_b$ . Because the of the transistors can vary a lot, it is a good design practice to make the  $R_b$  resistor about ½ the value shown in the above equation. Since the transistor is saturated, the increased base current produces the same  $V_{CE}$  and thus the same coil current.

Parameter	PN2222 ( $I_C=150mA$ )	2N2222 ( $I_C=500mA$ )	TIP120 ( $I_C=3A$ )
$h_{fe}$	100	40	1000
$h_{ie}$	60 $\Omega$	250 to 8000 $\Omega$	70 to 7000 $\Omega$
$V_{BEsat}$	0.6	2	2.5 V
$V_{CE}$ at saturation	0.3	1	2 V

**Table 10.2. Design parameters for the 2N2222 and TIP120.**

The IRF540 MOSFET is a voltage-controlled device, if the **Port** output is high, the 2N2222 is on, the MOSFET is off, and the coil current will be zero. If the **Port** output is low, the 2N2222 is off, the gate voltage of the MOSFET will be +V, the MOSFET is on, and the  $V_{DS}$  will be very close to 0. The IRF540 needs a large gate voltage ( $> 10V$ ) to fully turn so the drain will be able to sink up to 28 A.

Because of the resistance of the coil, there will not be significant  $dI/dt$  when the device is turned on. Consider a DC motor as shown in Figure 10.2 with  $V = 12\text{V}$ ,  $R = 50\ \Omega$  and  $L = 100\ \mu\text{H}$ . Assume we are using a 2N2222 with a  $V_{CE}$  of 1 V at saturation. Initially the motor is off (no current to the motor). At time  $t=0$ , the digital port goes from 0 to +3.3 V, and transistor turns on. Assume for this section, the emf is zero (motor has no external torque applied to the shaft) and the transistor turns on instantaneously, we can derive an equation for the motor ( $I_c$ ) current as a function of time. The voltage across both  $LC$  together is  $12 - V_{CE} = 11\ \text{V}$  at time  $= 0^+$ . At time  $= 0^+$ , the inductor is an open circuit. Conversely, at time  $= \infty$ , the inductor is a short circuit. The  $I_c$  at time  $0^-$  is 0, and the current will not change instantaneously because of the inductor. Thus, the  $I_c$  is 0 at time  $= 0^+$ . The  $I_c$  is  $11\text{V}/50\Omega = 220\text{mA}$  at time  $= \infty$ .

$$11\ \text{V} = I_c * R + L * d I_c / dt$$

General solution to this differential equation is

$$I_c = I_0 + I_1 e^{-t/T} \quad d I_c / dt = - (I_1 / T) e^{-t/T}$$

We plug the general solution into the differential equation and boundary conditions.

$$11\ \text{V} = (I_0 + I_1 e^{-t/T}) * R - L * (I_1 / T) e^{-t/T}$$

To solve the differential equation, the time constant will be  $T = L/R = 2\ \mu\text{sec}$ . Using initial conditions, we get

$$I_c = 220\text{mA} * (1 - e^{-t/2\mu\text{s}})$$

**Example 10.4.** Design an interface for two +12V 1A geared DC motors. These two motors will be used to propel a robot with two independent drive wheels.

**Solution:** We will use two copies of the TIP120 circuit in Figure 10.6 because the TIP120 can sink at least three times the current needed for this motor. We select a +12V supply and connect it to the +V in the circuit. The needed base current is

$$I_b = I_{coil} / h_{fe} = 1\text{A} / 1000 = 1\text{mA}$$

The desired interface resistor.

$$R_b \leq (V_{OH} - V_{be}) / I_b = (3.3 - 2.5) / 1\text{mA} = 800\ \Omega$$

To cover the variability in  $h_{fe}$ , we will use a 330  $\Omega$  resistor instead of the 800  $\Omega$ . The actual voltage on the motor when active will be  $+12 - 2 = 10\text{V}$ . The coils and transistors can vary a lot, so it is appropriate to experimentally verify the design by measuring the voltages and currents. Two PWM outputs are used to control the robot. The period of the PWM output is chosen to be about 10 times shorter than the time constant of the motor. The electronic driver will turn on and off at this rate, but the motor only responds to the average level. The software sets the duty cycle of the PWM to adjust the delivered power. When active, the interface will drive +10 V across the motor. The current will be a function of the friction applied to the shaft.

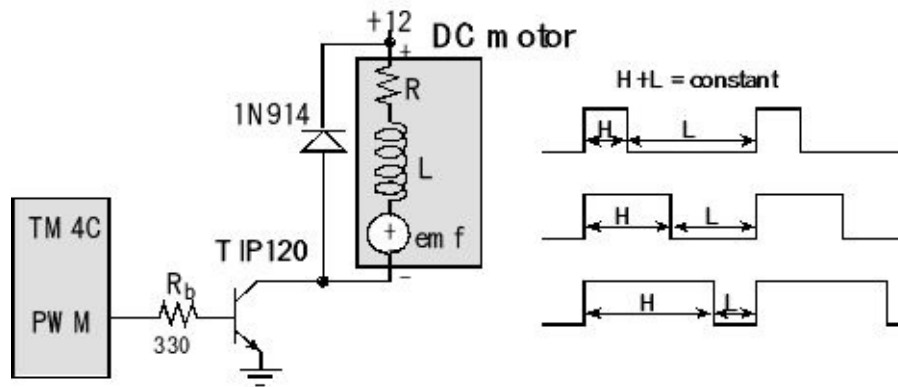


Figure 10.6. DC motor interface.

Similar to the solenoid and EM relay, the DC motor has a frame that remains motionless (called the **stator**), and an armature that moves (called the **rotor**), see Figure 10.7.

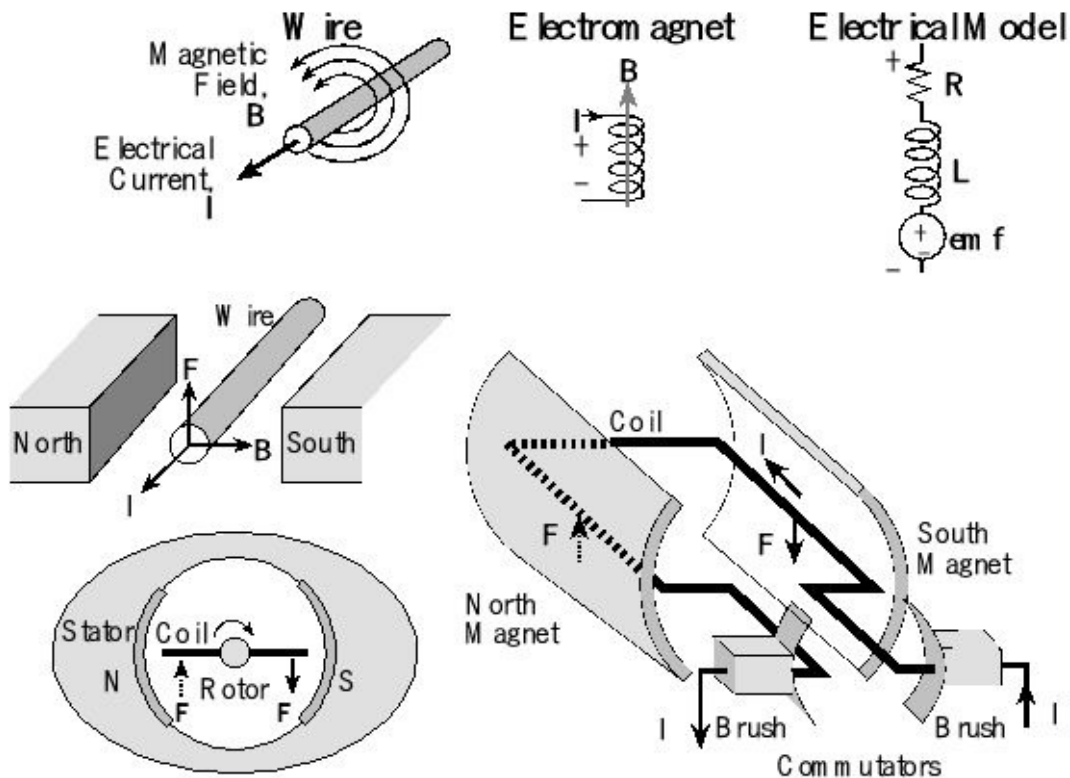


Figure 10.7. A brushed DC motor uses a commutator to flip the coil current.

A **brushed DC motor** has an electromagnetic coil as well, located on the rotor, and the rotor is positioned inside the stator. In Figure 10.7, North and South refer to a permanent magnet, generating a constant  $B$  field from left to right. In this case, the rotor moves in a circular manner. When current flows through the coil, a magnetic force is created causing a rotation of the shaft. A brushed DC motor uses commutators to flip the direction of the current in the coil. In this way, the coil on the right always has an up force, and the one on the left always has a down force. Hence, a constant current generates a continuous rotation of the shaft. When the current is



removed, the magnetic force stops, and the shaft is free to rotate. In a pulse-width modulated DC motor, the computer activates the coil with a current of fixed magnitude but varies the duty cycle in order to adjust the power delivered to the motor.

## 10.3. Sensors

Tachometers can be used to measure rotational speed of a motor. Some tachometers produce a sine wave with a frequency and amplitude proportional to motor speed. To use input capture, we need to convert the sine wave into a corresponding square wave of the same period. We can use a voltage comparator to detect events in an analog waveform. The input voltage range is determined by the analog supply voltages of the comparator. The output is takes on two values, shown as  $V_h$  and  $V_l$  in Figure 10.8. To reduce noise, a comparator with hysteresis has two thresholds,  $V_{t+}$  and  $V_{t-}$ . In both the positive and negative logic cases the threshold ( $V_{t+}$  or  $V_{t-}$ ) depends on the present value of the output.

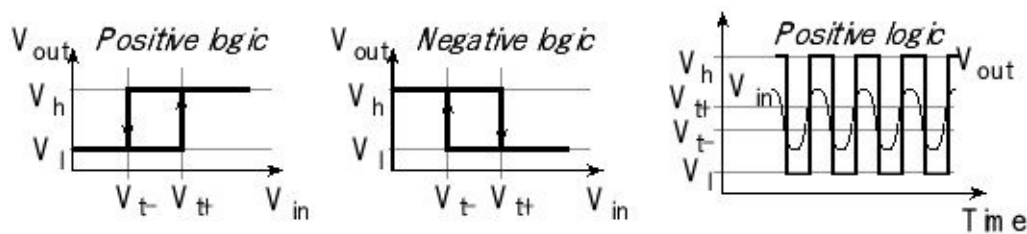


Figure 10.8. Input/output response of voltage converters with hysteresis.

Hysteresis prevents small noise spikes from creating a false trigger.

**Performance Tip:** In order to eliminate false triggering, we select a hysteresis level ( $V_{t+} - V_{t-}$ ) greater than the noise level in the signal.

In Figure 10.9, a rail-to-rail op amp is used to design a voltage comparator. Since the output swings from 0 to 3.3 V, it can be connected directly to an input pin of the microcontroller. On the other hand, since +3.3 and 0 are used to power the op amp, the analog input must remain in the 0 to +3.3 V range. The hysteresis level is determined by the amplitude of the output and the  $R_1/(R_1+R_2)$  ratio. If the output is at 0V, the voltage at the +terminal is  $V_{in} * R_2 / (R_1 + R_2)$ . The output switches when the voltage at the +terminal goes above 1.65. By solving for  $V_{in} * 200k / (10k + 200k) = 1.65$ , we see  $V_{in}$  must go above +1.73 for the output to switch. Similarly, if the output is at +3.3 V, the voltage at the +terminal can be calculated as  $V_{in} + (3.3 - V_{in}) * R_1 / (R_1 + R_2)$ . The output switches back when the voltage at the +terminal goes below 1.65. By solving for  $V_{in} + (3.3 - V_{in}) * R_1 / (R_1 + R_2) = 1.65$ , we see  $V_{in}$  go below +1.57 before the +terminal of the op amp falls below 1.65 V. In linear mode circuits we should not use the supply voltage to create voltage references, but in a saturated mode circuit, power supply ripple will have little effect on the response.

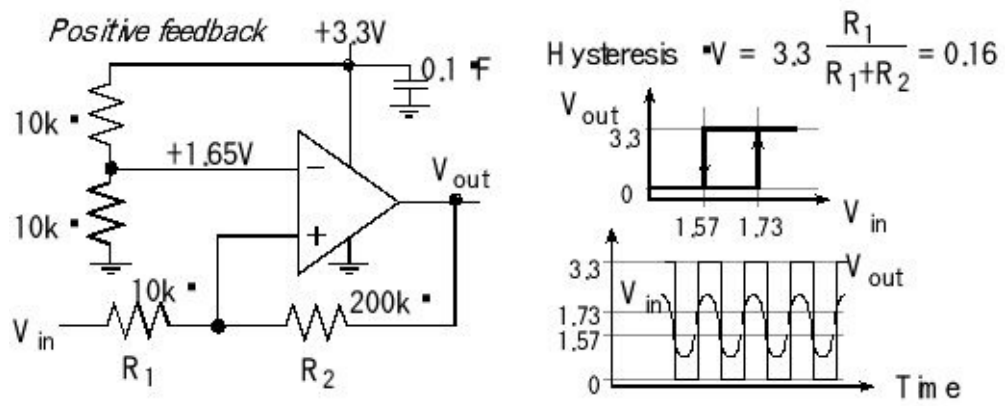


Figure 10.9. A voltage comparator with hysteresis using a rail to rail op amp.

## 10.4. Odometry

**Odometry** is a method to predict position from wheel rotations. We assume the wheels do not slip along the ground. If one wheel moves but the other does not, it will rotate about a single contact point of the wheel to the ground. If one wheel moves more than the other, then there will be both a motion and a rotation about a point somewhere along line defined by the axle connecting the two wheels. We define the robot center of gravity (cog) as a point equidistant from the pivot points. The robot position is defined as the  $(x,y)$  location and the compass direction, or yaw angle  $\theta$ , of the cog. See Figure 10.10.

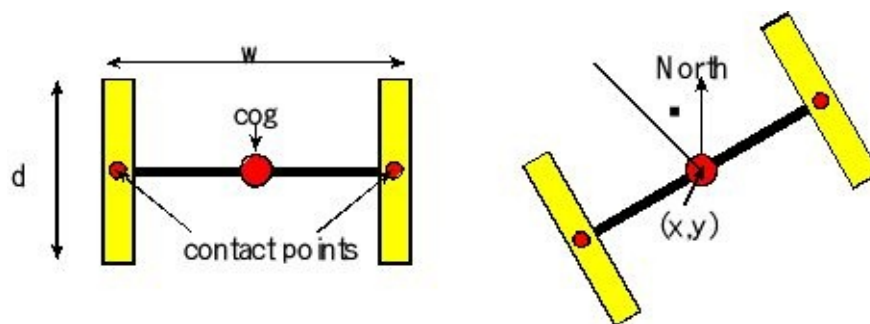


Figure 10.10. A robot with two drive wheels is defined by the wheel base and wheel diameter.

### Constants

Number of slots/rotation,  $n=32$

Wheel diameter,  $d = 886$  (0.01cm)

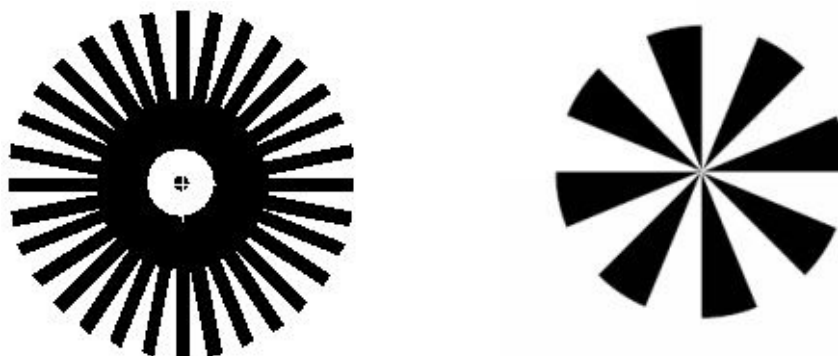


Figure 10.11. To measure wheel motion we used an encoder on each wheel.

Wheelbase (distance between wheels),  $w = 1651$  (0.01cm)

Wheel circumference,  $c = \pi d = 2783$  (0.01cm)

### Measurements

$L_{Count}$  the number of slots of left wheel in 349.5ms.  $R_{Count}$  the number of slots of right

wheel in 349.5ms. At 150 RPM, there will be 28 counts in 349.5 ms. Some simple cases are found in Table 10.3, where  $m$  is any number from -28 to +28.

Some

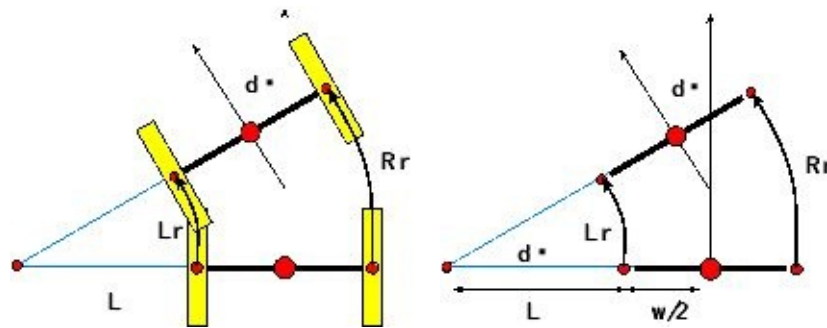
$LCount$	$RCount$	Motion
$m$	$m$	straight line motion in the current direction
0	$m$	pivot about stopped left motor
$m$	0	pivot about stopped right motor
$m$	$-m$	pure rotation about cog

**Table 10.3. Example measurements, relationship between counts and motion.**

**Derivations**

$Lr = L_{Count} * c/n$  the arc distance traveled by the left wheel (0.01cm)

$Rr = R_{Count} * c/n$  the arc distance traveled by the right wheel (0.01cm)



*Figure 10.12. Motions occurring during a left turn.*

Using similar triangles, we can find the new pivot point. Assuming  $Rr$  and  $Lr$  are both positive and  $Rr > Lr$ , we get

$$L/Lr = (L+w)/Rr$$

$$L/Lr - L/Rr = w/Rr$$

$$L Rr - L Lr = w Lr$$

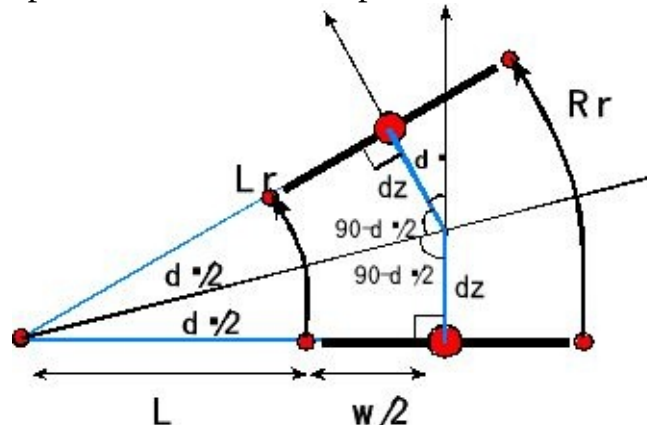
$$L = w Lr / (Rr - Lr)$$

Notice also the change in yaw,  $d\theta$ , is the same angle as the sector created by the change in axle.

The change in angle is

$$d\theta = Lr/L = Rr/(L+w)$$

We can divide the change in position into two components



### Figure 10.13. Geometry of a left turn.

The exact calculation for position change is

$$dz = (L+w/2)*\tan(d\theta/2)$$

but if  $d\theta$  is small, we can approximate  $dz$  by the arc length.

$$dz = d\theta/2*(L+w/2)$$

#### Initialize

We initialize the system by specifying the initial position and yaw.

$$(x, y, \theta) \quad (0.01\text{cm}, 0.01\text{cm}, 0.01 \text{radian})$$

**Calculations** (run this periodically, measuring  $L_{Count}$   $R_{Count}$ )

$$Lr = L_{Count} * c/n \quad (0.01\text{cm})$$

$$Rr = R_{Count} * c/n \quad (0.01\text{cm})$$

$$L = (w*Lr)/(Rr - Lr) \quad (0.01\text{cm})$$

$$d\theta = (100*Lr)/L \quad (0.01 \text{radian})$$

$$dz = ((d\theta/2)*(L+w/2))/100 \quad (0.01\text{cm}) \text{ approximation}$$

$$\text{or } dz = (\tan(d\theta/2)*(L+w/2))/100 \quad (0.01\text{cm}) \text{ more accurate}$$

$$x = x + dz*\cos(\theta) \quad (0.01\text{cm})$$

$$y = y + dz*\sin(\theta) \quad (0.01\text{cm}) \text{ first part of move}$$

$$\theta = \theta + d\theta \quad (0.01 \text{radian})$$

$$x = x + dz*\cos(\theta) \quad (0.01\text{cm})$$

$$y = y + dz*\sin(\theta) \quad (0.01\text{cm}) \text{ second part of move}$$

## 10.5. Simple Closed-Loop Control Systems.

A **bang-bang controller** uses a binary actuator, meaning the microcontroller output can be on or off. Other names for this controller are **binary controller**, **two-position controller**, and **on/off controller**. It is a closed-loop control system, because there is a sensor that measures the status of the system. This signal is called the measurand or state variable. Assume when the actuator is on the measurand increases, and when the actuator is off, the measurand decreases. There is a desired point for the measurand. The bang-bang controller is simple. If the measurand is too small, the actuator is turned on, and if the measurand is too large the actuator is turned off.

This digital control system applies heat to the room in order to maintain the temperature as close to the desired temperature as possible (Figure 10.14). This is a closed-loop control system because the control signals (heat) depend on the state variables (temperature). In this application, the actuator has only two states: *on* that warms up the room and *off* that does not apply heat. For this application to function properly, there must be a passive heat loss that lowers the room temperature when the heater is turned off. On a hot summer day, this heater system will not be able to keep the house cool. A bang-bang controller turns on the power if the measured temperature is too low and turns off the power if the temperature is too high. To implement **hysteresis**, we need two set-point temperatures,  $T_{high}$  and  $T_{low}$ . The controller turns on the power (activate relay) if the temperature goes below  $T_{low}$  and turns off the power (deactivate relay) if the temperature goes above  $T_{high}$ . The difference  $T_{high} - T_{low}$  is called hysteresis. The hysteresis extends the life of the relay by reducing the number of times the relay opens and closes.

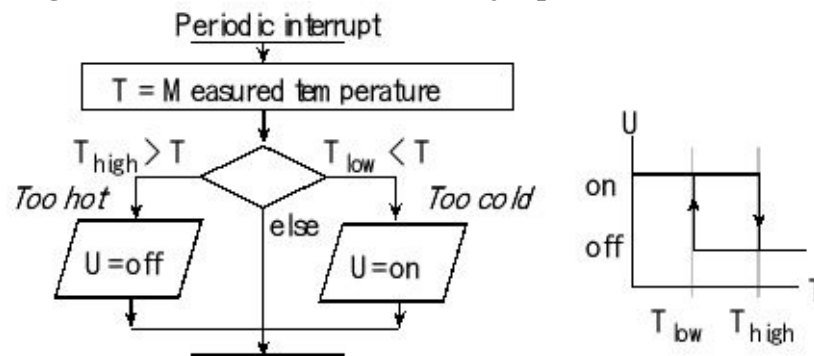


Figure 10.14. Flowchart of a Bang-Bang Temperature Controller

Assume the function **SE ()** returns the estimated temperature as a binary fixed-point number with a resolution of 0.5 °C. Program 10.1 uses a periodic interrupt so that the bang-bang controller runs in the background. The interrupt period is selected to be about the same as the time constant of the physical plant. The temperature variables **Tlow**, **Thigh** and **T** could be in any format, as long as the three formats are the same.

**Checkpoint 10.2:** What happens if **Tlow** and **Thigh** are too close together? What

happens if **Tlow** and **Thigh** are too far apart?

**Observation:** Bang-bang control works well with a physical plant with a very slow response.

```
int32_t Tlow,Thigh; // controller set points, 0.5 C
void Timer0A_Handler(void){
int32_t T=SE(); // estimated temperature, 0.5 C
  if(T < Tlow){
    TurnOn(); // too cold so turn on heat
  } else if (T > Thigh){
    TurnOff(); // too hot so turn off heat
  } // leave as is if Tlow<T<Thigh
  TIMER0_ICR_R = 0x01;// acknowledge timer0A periodic timer
}
```

*Program 10.1. Bang-bang temperature control software.*

An **incremental control system** uses an actuator with a finite number of discrete output states. For example, the actuator might be a PWM output with 249 possibilities from 2, 3, 4, ... 249 (0 to 100%). It is a closed-loop control system, because there is a sensor that measures the state variable. Assume when the actuator increases the measurand increases, and when the actuator decreases, the measurand decreases. There is a desired point for the measurand. The incremental controller is simple. If the measurand is too small, the actuator is increased, and if the measurand is too large, the actuator is decreased. It is important to choose the rate to run the controller properly. A good rule of thumb is to run the controller about 10 times faster than the time constant of the plant. The control system should make sure the actuator signal remains in the appropriate range. E.g., you do not want to increment an actuator output of 255 and get 0! The incremental controller is usually slow, but it has good accuracy and is very stable.

The objective of this incremental control system is to control the speed,  $X$ , of a DC motor shown in Figure 10.15. The actuator uses PWM to apply variable power to the motor. A tachometer is used to measure speed,  $X'$ .



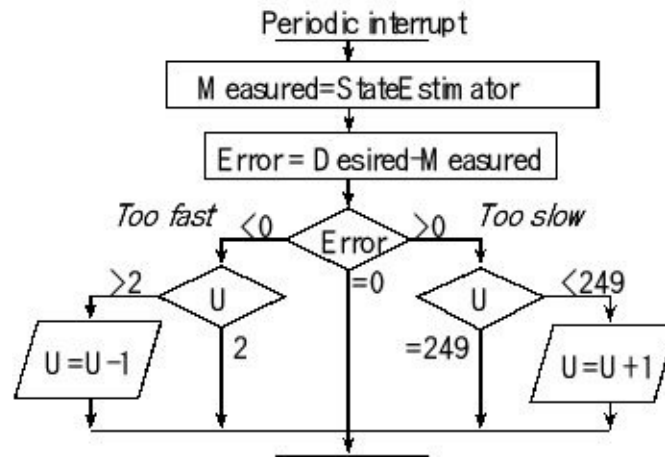


Figure 10.15. Flowchart of a position controller implemented using incremental control.

An incremental control algorithm simply adds or subtracts a constant from  $U$  depending on the sign of the error. In other words, if  $X$  is too slow then  $U$  is incremented and if  $X$  is too fast then  $U$  is decremented. It is important to choose the proper rate at which the incremental control software is executed. If it is executed too many times per second, then the actuator will saturate resulting in a Bang-Bang system. If it is not executed often enough then the system will not respond quickly to changes in the physical plant or changes in  $X^*$ . In this incremental controller we add or subtract "1" from the actuator, but a value larger than "1" would have a faster response at the expense of introducing oscillations.

**Common error:** An error will occur if the software does not check for overflow and underflow after  $U$  is changed.

**Observation:** If the incremental control algorithm is executed too frequently, then the resulting system behaves like a simple bang-bang controller.

**Observation:** Many control systems operate well when the control equations are executed about 10 times faster than the step response time of the physical plant.

Assume the function  $SE()$  returns measured speed. Program 10.2 uses a periodic interrupt so that the incremental controller runs in the background. The interrupt period is selected to be about 10 times smaller than the time constant of the physical plant. The optimal controller rate depends on the significance of the  $\pm 1$  value added to  $U$ . Experimental testing may be required to select an optimal controller rate, trading off response time for stability. Even though the position variables  $X$  and  $X^*$  may be unsigned, the error calculation  $E$  will be signed.

```

int32_t X,Xstar,E;    // speed, fixed-point in the same format
int32_t U;
void Timer0A_Handler(void){
    X = SE();        // estimated speed
    E = Xstar-X;    // error
    if(E < -10)    U--; // decrease if too fast
  }
  
```

```

else if(E > 10) U++; // increase if too slow
           // leave as is if close enough
if(U<2) U=2;    // underflow (minimum PWM)
if(U>249) U=249; // overflow (maximum PWM)
PWM0A_Duty(U); // output to actuator, Section 2.8
TIMER0_ICR_R = 0x01; // acknowledge timer0A periodic timer
}

```

*Program 10.2. Incremental control software for a DC motor.*

**Checkpoint 10.3:** In what ways would the controller behave differently if -10 and +10 were to be changed to 0 ?

**Checkpoint 10.4:** What happens if the interrupt period is too small (i.e., executes too frequently)?

**Observation:** It is a good debugging strategy to observe the assembly listing generated by the compiler when performing calculations on variables of mixed types (signed/unsigned, char/short).

**Observation:** Incremental control will work moderately well (accurate and stable) for an extremely wide range of applications. Its only short-coming is that the controller response time can be quite slow.

## 10.6. PID Controllers

### 10.6.1. General Approach to a PID Controller

The simple controllers presented in the last section are easy to implement, but will have either large errors or very slow response times. In order to make a faster and more accurate system, we can use linear control theory to develop the digital controller. There are three components of a proportional integral derivative **PID controller**.

$$U(t) = K_p E(t) + \int_0^t K_i E(\tau) d\tau + K_d \frac{dE(t)}{dt}$$

The error,  $E(t)$ , is defined as the present set-point,  $X^*(t)$ , minus the measured value of the controlled variable,  $X'(t)$ . See Figure 10.16.

$$E(t) = X^*(t) - X'(t)$$

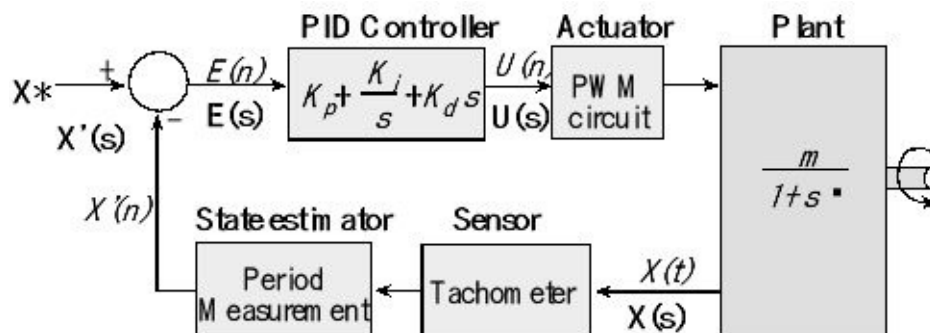


Figure 10.16. Block diagram of a linear control system in the frequency domain.

The PID controller calculates its output by summing three terms. The first term is proportional to the error. The second is proportional to the integral of the error over time, and the third is proportional to the rate of change (first derivative) of the error term. The values of  $K_p$ ,  $K_i$  and  $K_d$  are design parameters and must be properly chosen in order for the control system to operate properly. The proportional term of the PID equation contributes an amount to the control output that is directly proportional to the current process error. The gain term  $K_p$  adjusts exactly how much the control output response should change in response to a given error level. The larger the value of  $K_p$ , the greater the system reaction to differences between the set-point and the actual state variable. However, if  $K_p$  is too large, the response may exhibit an undesirable degree of oscillation or even become unstable. On the other hand, if  $K_p$  is too small, the system will be slow or unresponsive. An inherent disadvantage of proportional-only control is its inability to eliminate the steady state errors (offsets)

that occur after a set-point change or a sustained load disturbance.

The integral term converts the first order proportional controller into a second order system capable of tracking process disturbances. It adds to the controller output a factor that takes corrective action for any changes in the load level of the system. This integral term is scaled to the sum of all previous process errors in the system. As long as there is a process error, the integral term will add more amplitude to the controller output until the sum of all previous errors is zero. Theoretically, as long as the sign of  $K_i$  is correct, any value of  $K_i$  will eliminate offset errors. But, for extremely small values of  $K_i$ , the controlled variables will return to the set-point very slowly after a load upset or set-point change occurs. On the other hand, if  $K_i$  is too large, it tends to produce oscillatory response of the controlled process and reduces system stability. The undesirable effects of too much integral action can be avoided by proper tuning (adjusting) the controller or by including derivative action which tends to counteract the destabilizing effects.

The derivative action of a PID controller adds a term to the controller output scaled to the slope (rate of change) of the error term. The derivative term “anticipates” the error, providing a greater control response when the error term is changing in the wrong direction and a dampening response when the error term is changing in the correct direction. The derivative term tends to improve the dynamic response of the controlled variable by decreasing the process setting time, the time it takes the process to reach steady state. But if the process measurement is noisy, that is, if it contains high-frequency random fluctuations, then the derivative of the measured (controlled) variable will change wildly, and derivative action will amplify the noise unless the measurement is filtered.

**Checkpoint 10.5:** What happens in a PID controller if the sign of  $K_i$  is incorrect?

We can also use just some of the terms. For example a proportional/integrator (PI) controller drops the derivative term. We will analyze the digital control system in the frequency domain. Let  $X(s)$  be the Laplace transform of the state variable  $x(t)$ . Let  $X^*(s)$  be the Laplace transform of the desired state variable  $x^*(t)$ . Let  $E(s)$  be the Laplace transform of the error

$$E(s) = X^*(s) - X(s)$$

Let  $G(s)$  be the transfer equation of the PID linear controller. PID controllers are unique in this aspect. In other words, we cannot write a transfer equation for a bang-bang, incremental or fuzzy logic controller.

$$G(s) = K_p + K_d s + \frac{K_i}{s}$$

Let  $H(s)$  be the transfer equation of the physical plant. If we assume the physical plant (e.g., a DC motor) has a simple single pole behavior, then we can specify its response in the frequency domain with two parameters.  $m$  is the DC gain and  $t$  is its time constant. The transfer function of this simple motor is

$$H(s) = m/(1+ts)$$

The overall gain of the control system is

$$\frac{X(s)}{X^*(s)} = \frac{G(s)H(s)}{1+G(s)H(s)}$$

Theoretically we can choose controller constants,  $K_p$ ,  $K_i$  and  $K_d$ , to create the desired controller response. Unfortunately it can be difficult to estimate  $m$  and  $t$ . If a load is applied to the motor, then  $m$  and  $t$  will change.

To simplify the PID controller implementation, we break the controller equation into separate proportion, integral and derivative terms. I.e., let

$$U(t) = P(t) + I(t) + D(t)$$

where  $U(t)$  is the actuator output, and  $P(t)$ ,  $I(t)$  and  $D(t)$  are the proportional, integral and derivative components respectively. The proportional term makes the actuator output linearly related to the error. Using a proportional term creates a control system that applies more energy to the plant when the error is large. To implement the proportional term, we simply convert it to discrete time.

$$P(t) = K_p E(t) \Rightarrow P(n) = K_p E(n)$$

where the index “ $n$ ” refers to the discrete time input of  $E(n)$  and output of  $P(n)$ .

**Observation:** In order to develop digital signal processing equations, it is imperative that the control system be executed on a regular and periodic rate.

**Common error:** If the sampling rate varies, then controller errors will occur.

The integral term makes the actuator output related to the integral of the error. Using an integral term often will improve the steady state error of the control system. If a small error accumulates for a long time, this term can get large. Some control systems put upper and lower bounds on this term, called anti-reset-windup, to prevent it from dominating the other terms. The implementation of the integral term requires the use of a discrete integral or sum. If  $I(n)$  is the current control output, and  $I(n-1)$  is the previous calculation, the integral term is simply

$$I(t) = \int_0^t K_i E(\tau) d\tau \Rightarrow I(n) = \sum_1^n K_i E(n) \Delta t = I(n-1) + K_i E(n) \Delta t$$

where  $\Delta t$  is the sampling rate of  $E(n)$ .

The derivative term makes the actuator output related to the derivative of the error. This term is usually combined with either the proportional and/or integral term to improve the transient response of the control system. The proper value of  $K_d$  will provide for a quick response to changes in either the set point or loads on the physical plant. An incorrect value may create an overdamped (very slow response) or an underdamped (unstable oscillations) response. There are a couple of ways to implement the discrete time derivative. The simple approach is

$$D(t) = K_d \frac{dE(t)}{dt} \Rightarrow D(n) = K_d \frac{E(n) - E(n-1)}{\Delta t}$$

In practice, this first order equation is quite susceptible to noise. Figure 10.17 shows a sequence of  $E(n)$  with some added noise. Notice that huge errors occur when the above equation is used to calculate derivative.

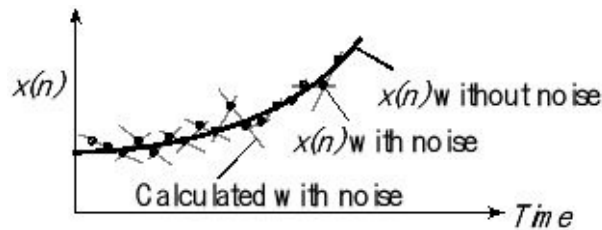


Figure 10.17. Illustration of the effect noise plays on the calculation of discrete derivative.

In most practical control systems, the derivative is calculated using the average of two derivatives calculated across different time spans. For example

$$D(n) = K_d \left( \frac{1}{2} \frac{E(n) - E(n-3)}{3\Delta t} + \frac{1}{2} \frac{E(n-1) - E(n-2)}{\Delta t} \right)$$

that simplifies to

$$D(n) = K_d \left( \frac{E(n) + 3E(n-1) - 3E(n-2) - E(n-3)}{6\Delta t} \right)$$

Linear regression through multiple points can yield the slope and yet be immune to noise.

**Checkpoint 10.6:** How is the continuous integral related to the discrete integral?

**Checkpoint 10.7:** How is the continuous derivative related to the discrete derivative?

## 10.6.2. Design Process for a PID Controller

The first design step is the analysis phase, where we determine specifications such as range, accuracy, stability, and response time for our proposed control system. A data acquisition system will be used to estimate the state variables. Thus, its range, accuracy and response time must be better than the desired specifications of the control system. We can use time-based techniques using input capture, or develop an ADC-based state estimator. In addition, we need to design an actuator to manipulate the state variables. It too must have a range and response time better than the controller specifications. The **actuator resolution** is defined as the smallest reliable change in output. For example, a 100 Hz PWM output generated by a 1  $\mu$ sec clock has 10,000 different outputs. For this actuator, the actuator resolution is  $\text{MaxPower}/10000$ . We wish to relate the actuator performance to the overall objective of controller accuracy. Thus, we need to map the effect on the state

variable caused a change in actuator output. This change in state variable should be less than or equal to the desired controller accuracy.

After the state estimator and actuator are implemented, the controller settings ( $K_p$ ,  $K_I$  and  $K_D$ ) must be adjusted so that the system performance is satisfactory. This activity is referred to as **controller tuning** or **field tuning**. If you perform controller tuning by guessing the initial setting then adjusting them by trial and error, it can be tedious and time consuming. Thus, it is desirable to have good initial estimates of controller settings. A good first setting may be available from experience with similar control loops. Alternatively, initial estimates of controller settings can be derived from the transient response of the physical plant. A simple open-loop method, called the **process reaction curve approach**, was first proposed by Ziegler/Nichols and Cohen/Coon in 1953. In this discussion, the term “process” as defined by Ziegler/Nichols means the same thing as the “physical plant” described earlier in this chapter. This open-loop method requires only that a single step input be imposed on the process. The process reaction method is based on a single experimental test that is made with the controller in the manual mode. A small step change,  $\Delta U$ , in the controller output is introduced and the measured process response is recorded, as shown in Figure 10.18. To obtain parameters of the process, a tangent is drawn to the process reaction curve at its point of maximum slope (at the inflection point). This slope is  $R$ , which is called the **process reaction rate**. The intersection of this tangent line with the original base line gives an indication of  $L$ , the process lag.  $L$  is really a measure of equivalent dead time for the process. If the tangent drawn at the inflection point is extrapolated to a vertical axis drawn at the time when the step was imposed, the amount by which this value is below the horizontal base line will be represented by the product  $L \cdot R$ .  $\Delta T$  is the time step for the digital controller. It is recommended to run P and PI controllers with  $\Delta T = 0.1L$ , and a PID controller at a rate 20 times faster ( $\Delta T = 0.05L$ .) Using these parameters, Ziegler and Nichol proposed initial controller settings as

#### **Proportional Controller**

$$K_p = \Delta U / (L \cdot R)$$

#### **Proportional-Integral Controller**

$$K_p = 0.9 \Delta U / (L \cdot R)$$

$$K_i = K_p / (3.33L)$$

#### **Proportional-Integral-Derivative Controller**

$$K_p = 1.2 \Delta U / (L \cdot R)$$

$$K_i = 0.5 K_p / L$$

$$K_d = 0.5 K_p L$$

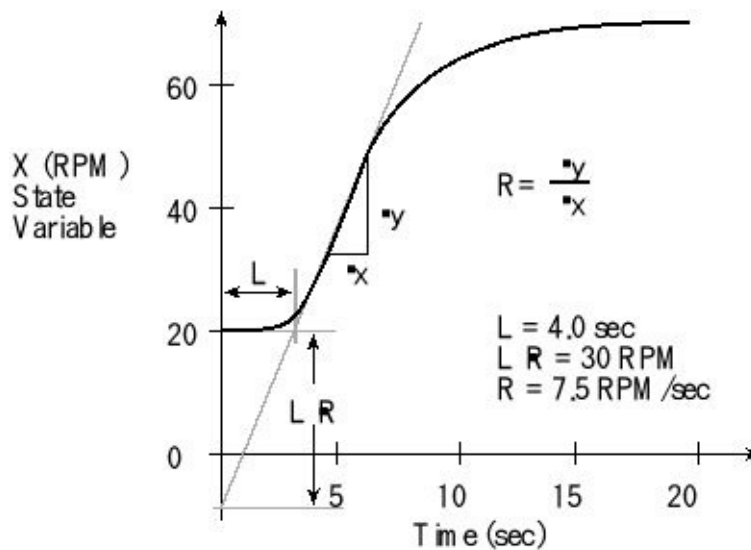


Figure 10.18. A process reaction curve used to determine controller settings.

**Checkpoint 10.8:** Are the Ziegler/Nichol equations consistent from a dimensional analysis perspective? In other words, are the units correct?

The **response time** is the delay after  $X^*$  is changed for the system to reach a new constant state. **Steady state controller accuracy** is defined as the average difference between  $X^*$  and  $X'$ . **Overshoot** is defined as the maximum positive error that occurs when  $X^*$  is increased. Similarly, **undershoot** is defined as the maximum negative error that occurs when  $X^*$  is decreased. During the testing phase, it is appropriate to add minimally intrusive debugging software that specifically measures performance parameters, such as response time, accuracy, overshoot, and undershoot. In addition, we can add instruments that allow us to observe the individual  $P(n)$ ,  $I(n)$  and  $D(n)$  components of the PID equation and their relation to controller error  $E(n)$ .

Once the initial parameters are selected, a simple empirical method can be used to fine-tune the controller. This empirical approach starts with proportional term ( $K_p$ ). As the proportional term is adjusted up or down, evaluate the quickness and smoothness of the controller response to changes in set-point and to changes in the load.  $K_p$  is too big if the actuator saturates both at the maximum and minimum after  $X^*$  is changed. The next step is to adjust the integral term ( $K_i$ ) a little at a time to improve the steady state controller accuracy without adversely affecting the response time. Don't change both  $K_p$  and  $K_i$  at once. Rather, you should vary them one at a time. If the response time, overshoot, undershoot and accuracy are within acceptable limits, then a PI controller is adequate. On the other hand, if accuracy and response are OK but overshoot and undershoot are unacceptable, adjust the derivative term ( $K_d$ ) to reduce the overshoots and undershoots.

We will design a proportional-integral motor control system. The overall objective is to control the speed of an object with an accuracy of 0.1 RPM and a range of 0 to 100



RPM as shown in Figure 10.1. Let  $X^*$  be the desired state variable. In this example,  $X^*$  will be a decimal fixed-point number and is set by the main program. Let  $X'$  be the estimated state variable that comes from the **state estimator**, which encodes the current position as the period of a squarewave, interfaced to an input capture pin. The period output of the sensor is linearly related to the position  $X$  with a fixed offset. The accuracy of the state estimator needs to match the 0.1 RPM specification of the controller. If  $p$  is the measured period in 0.1 ms and  $X'$  is the estimated speed in 0.1 RPM, the state estimator measures the period and calculates  $X'$ .

$$X' = p - 100$$

Let  $U$  be the actuator control variable ( $100 \leq U \leq 19900$ ). This system uses **pulse width modulation** with a 100 Hz squarewave that applies energy to the physical plant as shown in Figure 10.19.  $U$  will be the number of clock cycles (out of 20000) that the output is high. There is an external friction force slowing down on the motor. The PWM output from the computer creates a force causing the motor to spin faster.

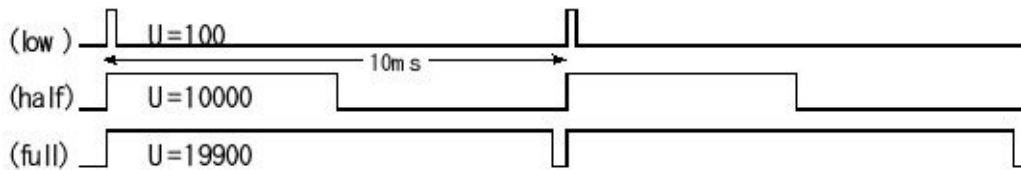


Figure 10.19. Pulse width modulated actuator signals.

The process reaction curve shown previously in Figure 10.18 was measured for this system after the actuator was changed from 250 to 2000, thus  $\Delta U$  is 1750 (units of clock cycles). From Figure 10.18, the lag  $L$  is 4.0 sec and the process reaction rate  $R$  is 7.5RPM/sec. The controller rate is selected to be about 10 times faster than the lag  $L$ , so  $\Delta T = 0.4$  sec. In this way, the controller runs at a rate faster than the physical plant. We calculate the initial PI controller settings using the Ziegler/Nichol equations.

$$K_p = 0.9 \Delta U / (L \cdot R) = 0.9 \cdot 1750 / (4.0 \cdot 7.5) = 52.5 \text{ cycles/RPM}$$

$$K_i = K_p / (3.33L) = 52.5 / (3.33 \cdot 4.0) = 3.94144 \text{ cycles/RPM/sec}$$

We will execute the PI control equation once every 0.4 second.  $X^*$  and  $X'$  are decimal fixed-point numbers with a resolution of 0.1 RPM. The constant 52.5 is expressed as 105/2. The extra divide by 10 handles the decimal fixed-point representation of  $X^*$  and  $X'$ .

$$P(n) = K_p \cdot (X^* - X') / 10 = 105 \cdot (X^* - X') / 20$$

We will also execute the integral control equation once every 0.4 second. Binary fixed-point is used to approximate 1.57658 as 101/64.

$$\begin{aligned} I(n) &= I(n-1) + K_i \cdot (X^* - X') \cdot \Delta T / 10 \\ &= I(n-1) + 3.94144 \cdot (X^* - X') \cdot 0.4 / 10 = I(n-1) + 101 \cdot (X^* - X') / 640 \end{aligned}$$

Program 10.3 shows an interrupt service handler, which runs at 10 kHz. The handler will establish the current **Time** in 0.1 ms. After 4000 interrupts (0.4 second), the control algorithm is implemented.

```
uint32_t Time; // Time in 0.1 msec
int32_t X;      // Estimated speed in 0.1 RPM, 0 to 1000
int32_t Xstar; // Desired speed in 0.1 RPM, 0 to 1000
int32_t E;      // Speed error in 0.1 RPM, -1000 to +1000
int32_t U,I,P;  // Actuator duty cycle, 100 to 19900 cycles
uint32_t Cnt;  // incremented every 0.1 msec
uint32_t Told; // used to measure period
void Timer0A_Handler(void){
    Time++;      // used to measure period
    if((Cnt++)==4000){ // every 0.4 sec
        Cnt = 0; // 0<X<100, 0<Xstar<100, 100<U<19900
        E = Xstar-X;
        P = (105*E)/20;
        I = I+(101*E)/640;
        if(I < -500) I=-500; // anti-reset windup
        if(I > 4000) I=4000;
        U = P+I; // PI controller has two parts
        if(U < 100) U=100; // Constrain actuator output
        if(U>19900) U=19900;
        PWM0A_Duty(U); // output to actuator, Section 2.8
    }
    TIMER0_ICR_R = 0x01; // acknowledge timer0A periodic timer
}
```

*Program 10.3. PI control software.*

**Checkpoint 10.9:** What is the output  $U$  of the controller if the speed  $X$  is much greater than the set-point  $X^*$ ? In this situation, what does the object do?

**Observation:** PID control will work extremely well (fast, accurate and stable) if the physical plant can be described with a set of linear differential equations.

---

## 10.7. Fuzzy Logic Control

There are a number of reasons to consider fuzzy logic approach to control. It is much simpler than PID systems. It will require less memory and execute faster. In other words, an 8-bit fuzzy system may perform as well (same steady state error and response time) as a 16-bit PID system. When complete knowledge about the physical plant is known, then a good PID controller can be developed. Since the fuzzy logic control is more robust (still works even if the parameter constants are not optimal), then the fuzzy logic approach can be used when complete knowledge about the plant is not known or can change dynamically. Choosing the proper PID parameters requires knowledge about the plant. The fuzzy logic approach is more intuitive, following more closely to the way a “human” would control the system. It is easy to modify an existing fuzzy control system into a new problem. The framework allows rapid prototyping.

Fuzzy logic was conceived in the mid-1960s by Lotfi Zadeh while at the University of California at Berkeley. However, the first commercial application didn't come until 1987, when the Matsushita Industrial Electric used it to control the temperature in a shower head. Named after the nineteenth-century mathematician George Boole, Boolean logic is an algebra where values are either true or false. This algebra includes operations of AND OR and NOT. Fuzzy logic is also an algebra, but where conditions may exist in the continuum between true and false. While Boolean logic defines two states, 8-bit fuzzy logic consists of 256 states all the way from “not at all” (0) to “definitely true” (255). “128” means half way between true and false. The fuzzy logic algebra also includes the operations of AND OR and NOT. A **fuzzy membership set**, a **fuzzy variable**, and a **fuzzy set** all refer to the same entity, which is a software variable describing the level of correctness for a condition within fuzzy logic. If we have a fuzzy membership set for the condition “hungry”, then as the value of hungry moves from 0 to 255, the condition “hungry” becomes more and more true.

....0.....32.....64.....96.....128.....160.....192.....224.....255  
Not at all ... a little bit ... somewhat ... mostly ... pretty much ...  
definitely

The design process for a fuzzy logic controller solves the following eight components. These components are listed in the order we would draw a data flow graph, starting with the state variables, progressing through the controller, and ending with the actuator output.

- The **Physical plant** has *real state variables*.
- The **Data Acquisition System** monitors these signals creating the *estimated state variables*.

- The **Preprocessor** may calculate relevant parameters called *crisp inputs*.
- **Fuzzification** will convert crisp inputs into *input fuzzy membership sets*.
- The **Fuzzy Logic** is a set of rules that calculate *output fuzzy membership sets*.
- **Defuzzification** will convert output sets into *crisp outputs*.
- The **Postprocessor** modify crisp outputs into a more convenient format.
- The **Actuator System** affects the Physical plant based on these output.

We will work through the concepts of fuzzy logic by considering examples of how we as humans control things like driving a car at a constant speed. During the initial stages of the design, we study the **physical plant** and decide which state variables to consider. For example, if we wish to control speed, then speed is obviously a state variable, but it might be also useful to know other forces acting on the object such as gravity (e.g., going up and down hills), wind speed and friction (e.g., rain and snow on the roadway). The purpose of the **data acquisition system** is to accurately measure the state variables. It is at this stage that the system converts physical signals into binary numbers to be processed by the software controller. We have seen two basic approaches in this book for this conversion: the measurement of period/frequency using input capture and the analog to digital conversion using an ADC. The **preprocessor** calculates **crisp inputs**, which are variables describing the input parameters in our software having units (like miles/hr). For example, if we measured speed, then some crisp inputs we might calculate would include speed error, and acceleration. Just like the PID controller, the accuracy of the data acquisition system must be better than the desired accuracy of the control system as a whole.

The next stage of the design is to consider the actuator and postprocessor. It is critical to be able to induce forces on the physical plant in a precise and fast manner. The step response of the actuator itself (time from software command to the application of force on the plant) must be faster than the step response of the plant (time from the application of force to the change in state variable.) Consider the case where we wish to control the temperature of a pot of water using a stove. The speed of the actuator is the time between turning the stove on and the time when heat is applied to the pot. The actuator on a gas stove is much faster than the actuator on an electric stove. The resolution of an actuator is the smallest change in output it can reliably generate. Just like the PID controller, the resolution of the actuator (converted into equivalent units on the input) must be smaller than the desired accuracy of the control system as a whole. A **crisp output** is a software variable describing the output parameters having units (like watts, Newtons, dynes/cm<sup>2</sup> etc.). The **postprocessor** converts the crisp output into a form that can be directly output to the actuator. The postprocessor can verify the output signals are within the valid range of the actuator. One of the advantages of fuzzy logic design is the usage of human intuition. Think carefully about how you control the actuator (gas pedal) when attempting to drive a car at a constant speed. There is no parameter in your brain specifying the exact position of the pedal (e.g., 50% pressed, 65% pressed etc.), unless of course you are city taxicab driver (where your brain allows two actuator states: full gas and full brake.) Rather, what your brain creates as actuator commands

are statements like “press the pedal little harder” and “press the pedal a lot softer.” So, the crisp output of fuzzy logic controller might be change in pedal pressure  $\Delta U$ , and the postprocessor would calculate  $U = U + \Delta U$ , then check to make sure  $U$  is within an acceptable range.

We continue the design of a fuzzy logic controller by analyzing its crisp inputs. As a design step, we create a list of true/false conditions that together describe the current state of the physical plant. In particular, we define **input fuzzy membership sets**, which are fuzzy logic variables describing conditions related to the state of the physical plant. These fuzzy variables do not need to be orthogonal. In other words, it is acceptable to have variables that are related to each other. When designing a speed controller, we could define multiple fuzzy variables referring to similar conditions, such as **WayTooFast**, **Fast**, and **LittleBitFast**. Given the scenario where we are driving too fast, there should be generous overlap in conditions, such that two or even three fuzzy sets are simultaneously partially true. On the other hand, it is important that the entire list of input membership fuzzy sets, when considered as an ensemble, form a complete definition of the status of the physical plant. For example, if we are attempting to drive a car at a constant speed, then **SlowingUp**, **GoingSteady**, and **SpeedingUp** might be input fuzzy variables describing the car’s acceleration. **Fuzzification** is the mathematical step converting the crisp inputs into input fuzzy membership sets. When implementing fuzzy logic explicitly with C code, we will have available the full set of AND, OR, NOT fuzzy logic operations.

The heart of a fuzzy logic controller is the **fuzzy logic** itself, which is set of logic equations that calculate fuzzy outputs as a function of fuzzy inputs. An **output fuzzy membership set** is a fuzzy logic variable describing a condition related to the actuator. **QuickStop**, **SlowDown**, **JustRight**, **MorePower**, and **MaxPower** are examples of output fuzzy variables that might be used to describe the action to perform on the gas pedal. Like input fuzzy variables, output fuzzy variables exist in the continuum from definitely false (0) to definitely true (1). Just like the input specification, it is also important to create a list of output membership fuzzy sets, when considered as an ensemble, form a complete characterization of what we wish to be able to do with the actuator. We write fuzzy logic equations using AND and OR functions in a way similar to Boolean logic. The fuzzy logic AND is calculated as the minimum value of the two inputs, and the fuzzy logic OR is calculated as the maximum value of the two inputs. The design of the rules, like the other aspects of fuzzy control, follows the human intuition.

**SlowDown = WayTooFast + SpeedingUp\*LittleBitFast**

**Checkpoint 10.10:** If **WayTooFast** is 50, **SpeedingUp** is 40, and **LittleBitFast** is 60, then what would be the calculated value for **SlowDown** ?

The **defuzzification** stage of the controller converts the output fuzzy variables into

crisp outputs. Although any function could be used, an effective approach is to use a weighted average. Consider the case where the pedal pressure  $U$  varies from 0 to 100, thus the crisp output  $\Delta U$  can take on values from -100 to +100. We think about what crisp output we want if just **QuickStop** were to be true. In this case, we wish to make  $\Delta U$  equal to -100. We then define crisp output values for **SlowDown**, **JustRight**, **MorePower**, and **MaxPower** as -10, 0, +10, and +100 respectfully. We can combine the five factors using a weighted average.

$$\Delta U = \frac{-100 * \text{QuickStop} - 10 * \text{SlowDown} + 10 * \text{MorePower} + 100 * \text{MaxPower}}{\text{QuickStop} + \text{SlowDown} + \text{JustRight} + \text{MorePower} + \text{MaxPower}}$$

Because the fuzzy controller is modular, we begin by testing each of the modules separately. The system-level testing of a fuzzy logic controller follows a procedure similar to the PID controller tuning. Debugging instruments can be added to record the crisp inputs, fuzzy inputs, fuzzy outputs, and crisp outputs during the real-time operation of the system. Fuzzification parameters are adjusted so that the status of the plant is captured in the set of values contained in the fuzzy input variables. Next, the rules are adjusted so that fuzzy output variables properly describe what we want to do with the actuator. Lastly, the defuzzification parameters are adjusted so the proper crisp outputs are created.

Next we will design a fuzzy logic motor controller. The actuator is a PWM (Figure 10.2). The power to the motor is controlled by varying the 8-bit PWM duty cycle. The motor speed is estimated with a tachometer connected to an input capture pin.

Our system has:

- two control inputs
  - $S^*$             the desired motor speed in RPM
  - $S'$             the current estimated motor speed RPM
- one control output
  - $N$             the digital value that we write to the PWM

To utilize 8-bit math, we change the units of speed to  $1000/256=3.90625$  RPM.

$$T^* = (256 \cdot S^*) / 1000 \quad \text{the desired motor speed in 3.9 RPM}$$

$$T' = (256 \cdot S') / 1000 \quad \text{the current estimated motor speed 3.9 RPM}$$

For example, if the desired speed is 500 RPM, then  $T^*$  will be 128. Notice that the estimated speed,  $T'$ , is measured by the input capture pin. In other words, the control system functions (estimate state variables, control equation calculations, and actuator output) are performed on a regular and periodic basis, every  $\Delta t$  time units. This allows signal processing techniques to be used. We will let  $T'(n)$  refer to the current measurement and  $T'(n-1)$  refer to the previous measurement, i.e., the one measured  $\Delta t$  time ago.

In the fuzzy logic approach, we begin by considering how a “human” would control the motor. Assume your hand were on a joystick (or your foot on a gas pedal) and consider how you would adjust the joystick to maintain a constant speed. We select

crisp inputs and outputs to base our control system on. It is logical to look at the error and the change in speed when developing a control system. Our fuzzy logic system will have two crisp inputs

$$E = T^* - T' \quad \text{the error in motor speed in 3.9rpm}$$

$$D = T'(n) - T'(n-1) \quad \text{the change in motor speed in 3.9rpm/time}$$

Notice that if we perform the calculations of  $D$  on periodic intervals, then  $D$  will represent the derivative of  $T'$ ,  $dT'/dt$ .  $T^*$  and  $T'$  are 8-bit unsigned numbers, so the potential range of  $E$  varies from -255 to +255. Errors beyond  $\pm 127$  will be adjusted to the extremes +127 or -128 without loss of information.

```
int8_t static Subtract(uint8_t N, uint8_t M){
// returns N-M
uint32_t N16,M16;
int32_t Result16;
    N16 = N;        // Promote N,M
    M16 = M;
    Result16 = N16-M16; // -255≤Result16≤+255
    if(Result16<-128) Result16 = -128;
    if(Result16>127) Result16 = 127;
    return(Result16);}

```

*Program 10.4. Subtraction with overflow/underflow checking.*

These are the global definitions of the input signals and fuzzy logic crisp input,

```
uint8_t Ts; // Desired Speed in 3.9 rpm units
uint8_t T; // Current Speed in 3.9 rpm units
uint8_t Told; // Previous Speed in 3.9 rpm units
int8_t D; // Change in Speed in 3.9 rpm/time units
int8_t E; // Error in Speed in 3.9 rpm units

```

*Program 10.5. Inputs and crisp inputs.*

**Common error:** Neglecting overflow and underflow can cause significant errors.

The need for the special **Subtract** function can be demonstrated with the following example:

**$E = T_s - T$ ; // if  $T_s=200$  and  $T=50$  then  $E$  will be -106!!**

This function can be used to calculate both  $E$  and  $D$ ,

```
void CrispInput(void){
    E = Subtract(Ts,T);
    D = Subtract(T,Told);
    Told = T;} // Set up Told for next time

```

*Program 10.6. Calculation of crisp inputs.*

Now, if  $T_s=200$  and  $T=50$  then  $E$  will be +127. To control the actuator, we could simply choose a new PWM value  $N$  as the crisp output. Instead, we will select,  $\Delta N$  that is the change in  $N$ , rather than  $N$  itself because it better mimics how a “human” would control it. Again, think about how you control the speed of your car when driving. You do not adjust the gas pedal to a certain position, but rather make small or large changes to its position in order to speed up or slow down. Similarly, when controlling the temperature of the water in the shower, you do not set the hot/cold controls to certain absolute positions. Again you make differential changes to affect the “actuator” in this control system. Our fuzzy logic system will have one crisp output:

$\Delta N$                     change in output,  $N=N+\Delta N$  in PWM units

Next we introduce fuzzy membership sets that define the current state of the crisp inputs and outputs. Fuzzy membership sets are variables that have true/false values. The value of a fuzzy membership set ranges from definitely true (255) to definitely false (0). For example, if a fuzzy membership set has a value of 128, you are stating the condition is half way between true and false. For each membership set, it is important to assign a meaning or significance to it. The calculation of the input membership sets is called **Fuzzification**. For this simple fuzzy controller, we will define 6 membership sets for the crisp inputs:

<i>Slow</i>	True if the motor is spinning too slow
<i>OK</i>	True if the motor is spinning at the proper speed
<i>Fast</i>	True if the motor is spinning too fast
<i>Up</i>	True if the motor speed is getting larger
<i>Constant</i>	True if the motor speed is remaining the same
<i>Down</i>	True if the motor speed is getting smaller.

We will define 3 membership sets for the crisp output:

<i>Decrease</i>	True if the motor speed should be decreased
<i>Same</i>	True if the motor speed should remain the same
<i>Increase</i>	True if the motor speed should be increased

The fuzzy membership sets are usually defined graphically, but software must be written to actually calculate each. In this implementation, we will define three adjustable thresholds,  $T_E$ ,  $T_D$  and  $T_N$ . These are software constants and provide some fine tuning to the control system. We will set each threshold to 20. If you build one of these fuzzy systems, try varying one threshold at a time and observe the system behavior (steady state controller error and transient response.) If the error,  $E$ , is -5 (3.9rpm units), the fuzzy logic will say that *Fast* is 64 (25% true), *OK* is 192 (75% true), and *Slow* is 0 (definitely false.) If the error,  $E$ , is +21 (in 3.9rpm units), the fuzzy logic will say that *Fast* is 0 (definitely false), *OK* is 0 (definitely false), and *Slow* is 255 (definitely true.)  $T_E$  is defined to be the error (e.g., 20 in 3.9 rpm units is 78 rpm) above which we will definitely consider the speed to be too fast.



Similarly, if the error is less than  $-TE$ , then the speed is definitely too slow.

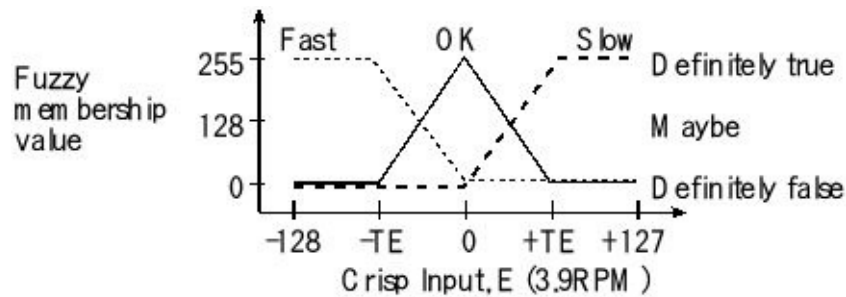


Figure 10.20. Fuzzification of the error input.

In this fuzzy system, the input membership sets are continuous piece-wise linear functions. Also, for each crisp input value, *Fast*, *OK*, *Slow* sum to 255. In general, it is possible for the fuzzy membership sets to be nonlinear or discontinuous, and the membership values do not have to sum to 255. The other three input fuzzy membership sets depend on the crisp input, *D*. *TD* is defined to be the change in speed (e.g., 20 in 3.9 rpm/time units is 78 rpm/time) above which we will definitely consider the speed to be going up. Similarly, if the change in speed is less than  $-TD$ , then the speed is definitely going down.

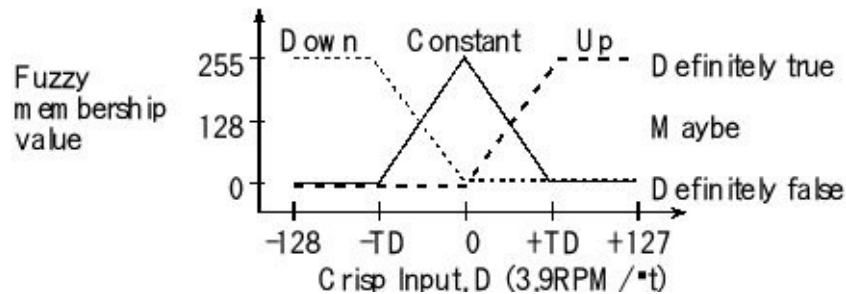


Figure 10.21. Fuzzification of the acceleration input.

In C, we could define a fuzzy function that takes the crisp inputs and calculates the fuzzy membership set values. Again *TE* and *TD* are software constants that will affect the controller error and response time.

```
#define TE 20
uint8_t Fast, OK, Slow, Down, Constant, Up;
#define TD 20
uint8_t Increase, Same, Decrease;
#define TN 20
void InputMembership(void){
    if(E <= -TE) {          // E ≤ -TE
        Fast = 255;
        OK = 0;
        Slow = 0;}
    else
        if(E < 0){          // -TE < E < 0
```

```

Fast = (255*(-E))/TE;
OK = 255-Fast;
Slow = 0;}
else
if(E < TE){      // 0<E<TE
  Fast = 0;
  Slow = (255*E)/TE;
  OK = 255-Slow;}
else {           // +TE≤E
  Fast = 0;
  OK = 0;
  Slow = 255;}
if(D <= -TD) {   // D≤-TD
  Down = 255;
  Constant = 0;
  Up = 0;}
else
if(D < 0){       // -TD<D<0
  Down = (255*(-D))/TD;
  Constant = 255-Down;
  Up = 0;}
else
if(D < TD){      // 0<D<TD
  Down = 0;
  Up = (255*D)/TD;
  Constant = 255-Up;}
else{           // +TD≤D
  Down = 0;
  Constant = 0;
  Up = 255;}
}

```

*Program 10.7. Calculation of the fuzzy membership variables in C.*

The fuzzy rules specify the relationship between the input fuzzy membership sets and the output fuzzy membership values. It is in these rules that one builds the intuition of the controller. For example, if the error is within reasonable limits and the speed is constant, then the output should not be changed. In fuzzy logic we write:

If *OK* and *Constant* then *Same*

If the error is within reasonable limits and the speed is going up, then the output should be reduced to compensate for the increase in speed. I.e.,

If *OK* and *Up* then *Decrease*

If the motor is spinning too fast and the speed is constant, then the output should be reduced to compensate for the error. I.e.,

If *Fast* and *Constant* then *Decrease*

If the motor is spinning too fast and the speed is going up, then the output should be reduced to compensate for both the error and the increase in speed. I.e.,

If *Fast* and *Up* then *Decrease*

If the error is within reasonable limits and the speed is going down, then the output should be increased to compensate for the drop in speed. I.e.,

If *OK* and *Down* then *Increase*

If the motor is spinning too slowly and the speed is constant, then the output should be increased to compensate for the error. I.e.,

If *Slow* and *Constant* then *Increase*

If the motor is spinning too slowly and the speed is going down, then the output should be increase to compensate for both the error and the drop in speed. I.e.,

If *Slow* and *Down* then *Increase*

These 7 rules can be illustrated in a table form.

		D		
		<i>Down</i>	<i>Constant</i>	<i>Up</i>
E	<i>Slow</i>	<i>Increase</i>	<i>Increase</i>	
	<i>OK</i>	<i>Increase</i>	<i>Same</i>	<i>Decrease</i>
	<i>Fast</i>		<i>Decrease</i>	<i>Decrease</i>

Figure 10.22. Fuzzy logic rules shown in table form.

It is not necessary to provide a rule for all situations. For example, we did not specify what to do if *Fast&Down* or for *Slow&Up*. Although we could have added (but did not):

If *Fast* and *Down* then *Same*

If *Slow* and *Up* then *Same*

When more than one rule applied to an output membership set, then we can combine the rules:

Same=(OKandConstant)

Decrease=(OKandUp)or(FastandConstant)or(FastandUp)

Increase=(OKandDown)or(SlowandConstant)or(SlowandDown)

In fuzzy logic, the **and** operation is performed by taking the minimum and the **or** operation is the maximum. Thus the C function that calculates the three output fuzzy membership sets is

```
uint8_t static min(uint8_t u1,uint8_t u2){
    if(u1>u2) return(u2);
    else return(u1);}
uint8_t static max(uint8_t u1,uint8_t u2){
    if(u1<u2) return(u2);
    else return(u1);}
void OutputMembership(void){
    Same    = min(OK,Constant);
    Decrease = min(OK,Up)
    Decrease = max(Decrease,min(Fast,Constant));
    Decrease = max(Decrease,min(Fast,Up));
    Increase = min(OK,Down)
    Increase = max(Increase,min(Slow,Constant));
    Increase = max(Increase,min(Slow,Down));}
```

Program 10.8. Calculation of the output fuzzy membership variables in C.

The calculation of the crisp outputs is called **Defuzzification**. The fuzzy membership sets for the output specifies the crisp output,  $\Delta N$ , as a function of the membership value. For example, if the membership set *Decrease* were true (255) and the other two were false (0), then the change in output should be  $-TN$  (where  $TN$  is another software constant). If the membership set *Same* were true (255) and the other two were false (0), then the change in output should be 0. If the membership set *Increase* were true (255) and the other two were false (0), then the change in output should be  $+TN$ .

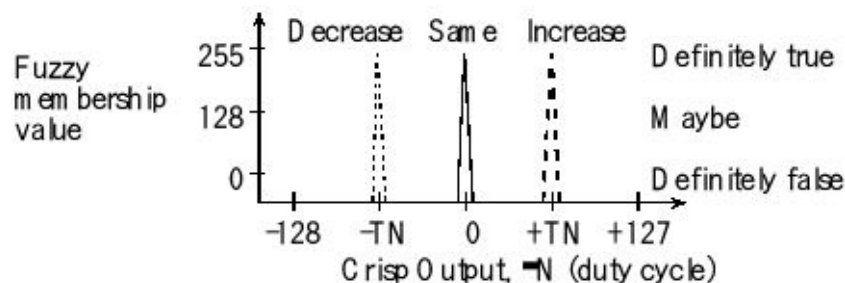


Figure 10.23. Defuzzification of the  $\Delta N$  crisp output.

In general, we calculate the crisp output as the weighted average of the fuzzy membership sets:

$$\Delta N = \frac{Decrease \cdot (-TN) + Same \cdot 0 + Increase \cdot TN}{Decrease + Same + Increase}$$

The C compiler will promote the calculations to 32 bits, and perform the calculation using 32-bit signed math that will eliminate overflow on intermediate terms. The output,  $dN$ , will be bounded in between  $-TN$  and  $+TN$ . Thus the C function that calculates the crisp output is

```
int32_t dN;
void CrispOutput(void){
    dN=(TN*(Increase-Decrease))/(Decrease+Same+Increase);
}
```

*Program 10.9. Calculation of the crisp output in C.*

```
void Timer0A_Handler(void){
    T = SE();          // estimate speed, set T, 0 to 255
    CrispInput();     // Calculate E,D and new Told
    InputMembership(); // Sets Fast,OK,Slow,Down,Constant,Up
    OutputMembership(); // Sets Increase,Same,Decrease
    CrispOutput();    // Sets dN
    N = max(0,min(N+dN,255));
    PWM0A_Duty(N);    // output to actuator, Section 2.8
    TIMER0_ICR_R = 0x01; // acknowledge timer0A periodic timer
}
```

*Program 10.10. Periodic interrupt service for fuzzy logic controller.*

**Observation:** Fuzzy logic control will work extremely well (fast, accurate and stable) if the designer has expert knowledge (intuition) of how the physical plant behaves.

---

## 10.8. Exercises

**10.1** For each term give a definition in 16 words or less.

- a) State variable      b) State estimator      c) Closed loop
- d) Transient response      e) Stability      f) Steady state accuracy
- g) Process reaction curve      h) Process reaction rate      i) Anti-reset windup

**10.2** For each control algorithm give a definition in 16 words or less.

- a) Open loop      b) Bang-bang      c) Incremental
- d) PID      e) Input PI      f) Fuzzy logic

**10.3** For each Fuzzy Logic term give a definition in 16 words or less.

- a) Crisp input      b) Fuzzification      c) Fuzzy membership set
- d) Fuzzy logic      e) Defuzzification      f) Crisp output

**10.4** Briefly explain why it is important to choose the proper update rate for a fuzzy logic controller. In particular, explain what happens to a fuzzy logic controller if the controller is executed too infrequently. Similarly, explain what happens to a fuzzy logic controller if the controller is executed too frequently.

**10.5.** Assume you have an 8-bit fuzzy logic system like the ones described in this chapter. Write formal descriptions for the complement and exclusive or fuzzy logic operations. Show C code implementations for these two functions.

**10.6** The objective of this problem is to use the Ziegler and Nichol approach to develop the PI controller equations that allow an embedded system to control a DC motor. The state variable is speed, which is measured using 16-bit input capture and has a measurement resolution of 1 RPM. The input capture device driver repeatedly updates a global variable, called **Speed**. This 16-bit unsigned variable has units of RPM and a range of 0 to 20000. The microcontroller uses pulse-width modulation to control power to the motor. The controller software writes to a global variable, called **Duty**, which ranges from 0 (0%) to 10000 (100%). The following plot shows an experimental measurement obtained when **Duty** is changed from 2500 to 5000. The desired speed is stored in the global variable, **Desired**, which has the same units as **Speed**. Design a fixed-point PI controller that takes **Speed** and **Desired** as inputs and calculates **Duty** as an output. From the response graph in Figure 10.24, estimate the **L** and **R** parameters of the Ziegler and Nichol method. How often should the controller be executed? Show just the equations (no software or hardware is required), calculating **Duty** as a function of **Speed** and **Desired**.

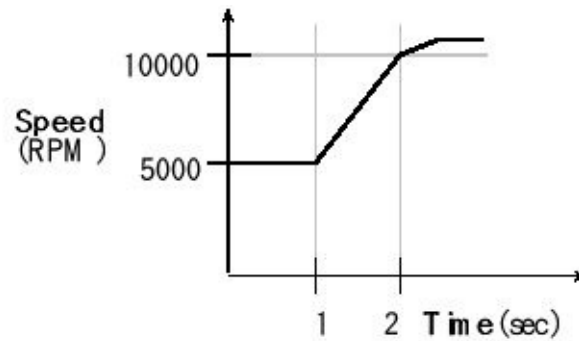


Figure 10.24. A process reaction curve for the DC motor.

**10.7** The objective of this problem is to use the Ziegler and Nichol approach to develop the PID controller equations that allow an embedded system to control the DC motor presented in Question 10.6. I.e., work through the steps of Question 10.6 for a PID system.

**10.8** Create a definition for Fuzzy Logic complement. Let  $\sim A$  be the complement of  $A$ . Some of these logic equations are valid for Fuzzy Logic and some are not. For each valid equation, present a formal proof of its correctness. For each invalid equation, give a counter example.

- a)  $A*B = B*A$
- b)  $A+B = B+A$
- c)  $(A*B)*C = A*(B*C)$
- d)  $(A+B)+C = A+(B+C)$
- e)  $(A+B)*C = (A*C)+(B*C)$
- f)  $A + \sim A = \text{true}$
- g)  $A * \sim A = \text{false}$
- h)  $(A*B)+A = A$

# Appendix 1. Glossary

**1/f noise** A fundamental noise in resistive devices arising from fluctuating conductivity. Same as pink noise.

**2's complement** (see two's complement).

**60 Hz noise** An added noise from electromagnetic fields caused by either magnetic field induction or capacitive coupling.

**accumulator** High-speed memory located in the processor used to perform arithmetic or logical functions. The accumulators on the ARM Cortex M are Registers R0 through R12.

**accuracy** A measure of how close our instrument measures the desired parameter referred to the NIST.

**acknowledge** Clearing the interrupt flag bit that requested the interrupt.

**active thread** A thread that is in the ready-to-run circular linked list. It is either running or is ready to run.

**actuator** Electro-mechanical or electro-chemical device that allows computer commands to affect the external world.

**ADC** Analog to digital converter, an electronic device that converts analog signals (e.g., voltage) into digital form (i.e., integers).

**address bus** A set of digital signals that connect the CPU, memory and I/O devices, specifying the location to read or write for each bus cycle. See also control bus and data bus.

**aging** A technique used in priority schedulers that temporarily increases the priority of low priority threads so they are run occasionally. (See starvation)

**aliasing** When digital values sampled at  $f_s$  contain frequency components above  $\frac{1}{2} f_s$ , then the apparent frequency of the data is shifted into the 0 to  $\frac{1}{2} f_s$  range. See Nyquist Theory.

**alternatives** The total number of possibilities. E.g., an 8-bit number scheme can represent 256 different numbers. An 8-bit digital to analog converter (DAC) can generate 256 different analog outputs.

**anode** The positive side of a diode. Current enters the anode side of a diode. Contrast with cathode.

**answer modem** The device that receives the telephone call.

**anti-reset-windup** Establishing an upper bound on the magnitude of the integral term in a PID controller, so this term will not dominate, when the errors are large.

**arithmetic logic unit (ALU)** Component of the processor that performs arithmetic and logic operations.

**arm** Activate so that interrupts are requested. Trigger flags that can request interrupts will have a corresponding arm bit to allow or disallow that flag to request interrupts. Contrast to enable.

**armature** The moving structure in a relay, the part that moves when the relay is activated. Contrast to frame.

**ASCII** American Standard Code for Information Interchange, a code for representing characters, symbols, and synchronization messages as 7 bit, 8-bit or 16-bit binary values.

**assembler** System software that converts an assembly language program (human readable format) into object code (machine readable format).



**assembly directive** Operations included in the program that are not executed by the computer at run time, but rather are interpreted by the assembler during the assembly process. Same as pseudo-op.

**assembly listing** Information generated by the assembler in human readable format, typically showing the object code, the original source code, assembly errors, and the symbol table.

**asynchronous bus** A communication protocol without a central clock where the data is transferred using two or three control lines implementing a handshaked interaction between the memory and the computer.

**asynchronous protocol** A protocol where the two devices have separate and distinct clocks

**atomic** Software execution that cannot be divided or interrupted. Once started, an atomic operation will run to its completion without interruption. Most assembly language instructions are atomic. All instructions on the Cortex-Mprocessor are atomic except store and load multiple, **STM LDM**.

**autoinitialization** The process of automatically reloading the address registers and block size counters at the end of a previous block transfer, so that DMA transfer can occur indefinitely without software interaction.

**availability** The portion of the total time that the system is working. MTBF is the mean time between failures, MTTR is the mean time to repair, and availability is  $MTBF/(MTBF+MTTR)$ .

**bandwidth** In communication systems, the information transfer rate, the amount of data transferred per second. Same as throughput. In analog circuits, the frequency at which the gain drops to 0.707 of the normal value. For a low pass system, the frequency response ranges from 0 to a maximum value. For a high pass system, the frequency response ranges from a minimum value to infinity. For a bandpass system, the frequency response ranges from a minimum to a maximum value. Compare to frequency response.

**bandwidth coupling** Module A is connected to Module B, because data flows from A to B.

**bang-bang** A control system where the actuator has only two states, and the system “bangs” all the way in one direction or “bangs” all the way in the other, same as binary controller.

**bank-switched memory** A memory module with two banks that interfaces to two separate address/data buses. At any given time one memory bank is attached to one address/data bus the other bank is attached to the other bus, but this attachment can be switched.

**basis** Subset from which linear combinations can be used to reconstruct the entire set.

**baud rate** In general, the baud rate is the total number of bits (information, overhead, and idle) per time that are transmitted. In a modem application it is the total number of sounds per time are transmitted

**bi-directional** Digital signals that can be either input or output.

**biendian** The ability to process numbers in both big and little-endian formats.

**big endian** Mechanism for storing multiple byte numbers such that the most significant byte exists first (in the smallest memory address). See also little endian.

**binary** A system that has two states, on and off.

**binary controller** Same as bang-bang.

**binary recursion** A recursive technique that makes two calls to itself during the execution

of the function. See also recursion, linear recursion, and tail recursion.

**binary semaphore** A semaphore that can have two values. The value=1 means OK and the value=0 means busy. Compare to counting semaphore.

**bipolar transistor** Either a NPN or PNP transistor.

**bipolar stepper motor** A stepper motor where the current flows in both directions (in/out) along the interface wires; a stepper with four interface wires. Contrast to unipolar stepper motor.

**bit** Basic unit of digital information taking on the value of either 0 or 1.

**bit rate** The information transfer rate, given in bits per second. Same as bandwidth and throughput.

**bit time** The basic unit of time used in serial communication. With serial channel bit time is 1/ baud rate.

**blind-cycle** A software/hardware synchronization method where the software waits a specified amount of time for the hardware operation to complete. The software has no direct information (blind) about the status of the hardware.

**block correction code (BCC)** A code (e.g., horizontal parity) attached to the end of a message used to detect and correct transmission errors.

**blocked thread** A thread that is not scheduled for running because it is waiting on an external event.

**blocking semaphore** A semaphore where the threads will block (so other threads can perform useful functions) when they execute wait on a busy semaphore. Contrast to spinlock semaphore.

**Bluetooth** A low-power, wireless personal area network that allows pairing, which is two devices communicating with each other.

**Board Support Package (BSP)** A set of software routines that abstract the I/O hardware such that the same high-level code can run on multiple computers.

**borrow** During subtraction, if the difference is too small, then we use a borrow to pass the excess information into the next higher place. For example, in decimal subtraction 36-27 requires a borrow from the ones to tens place because 6-7 is too small to fit into the 0 to 9 range of decimal numbers.

**bounded waiting** The condition where once a thread begins to wait on a resource, there are a finite number of threads that will be allowed to proceed before this thread is allowed to proceed.

**break-before-make** In a double-throw relay or double-throw switch, there is one common contact and two separate contacts. Break-before-make means as the common contact moves from one of separate contacts to another, it will break off (finish bouncing and no longer touch) the first contact before it makes (begins to bounce and starts to touch) the other contact. A *form C* relay has a *break-before-make* operation.

**break or trap** A break or a trap is a debugging instrument that halts the processor. With a resident debugger, the break is created by replacing specific op code with a software interrupt instruction. When encountered it will stop your program and jump into the debugger. Therefore, a break halts the software. The condition of being in this state is also referred to as a break.

**breakdown** A transducer that stops functioning when its input goes above a maximum value

or below a minimum value. Contrast to dead zone.

**breakpoint** The place where a break is inserted, the time when a break is encountered, or the time period when a break is active.

**brushed DC motor** A motor where the current reversals are produced with brushes between the stator and rotor. They are less expensive than brushless DC motors.

**brushless DC motor (BLDC)** A motor where the current reversals are produced with shaft sensors and an electronic controller. They are faster and more reliable than brushed DC motors.

**buffered I/O** A FIFO queue is placed in between the hardware and software in an attempt to increase bandwidth by allowing both hardware and software to run in parallel.

**burn** The process of programming a ROM, PROM or EEPROM.

**burst DMA** An I/O synchronization scheme that transfers an entire block of data all at once directly from an input device into memory, or directly from memory to an output device.

**bus** A set of digital signals that connect the CPU, memory and I/O devices, consisting of address signals, data signals and control signals. See also address bus, control bus and data bus.

**bus bandwidth** The number of bytes transferred per second between the processor and memory.

**bus interface unit (BIU)** Component of the processor that reads and writes data from the bus. The BIU drives the address and control buses.

**busy-wait synchronization** A software/hardware synchronization method where the software continuously reads the hardware status waiting for the hardware operation to complete. The software usually performs no work while waiting for the hardware. Same as gadfly.

**byte** Digital information containing eight bits.

**carrier frequency** the average or midvalue sound frequency in the modem.

**cathode** The negative side of a diode. Current exits the cathode side of a diode. Contrast to anode.

**causal** The property where the output depends on the present and past inputs, but not on any future inputs.

**ceiling** Establishing an upper bound on the result of an operation. See also floor.

**certification** A process where a governing body (e.g., FDA, NASA, FCC, DOD etc.) gives approval for the use of the device. It usually involves demonstrating the device meets or exceeds safety and performance criteria.

**channel** The hardware that allows communication to occur.

**characteristic** A Bluetooth functionalities that allows data to be exchanged.

**checksum** The simple sum of the data, usually in finite precision (e.g., 8, 16, 24 bits).

**closed-loop control system** A control system that includes sensors to measure the current state variables. These inputs are used to drive the system to the desired state.

**CMOS** A digital logic system called complementary metal oxide semiconductor. It has properties of low power and small size. Its power is a function of the number of transitions per second. Its speed is often limited by capacitive loading.

**cohesion** A cohesive module is one such that all parts of the module are related to each other to satisfy a common objective.

**common mode** For a system with differential inputs, the common mode properties are defined as signals applied to both inputs simultaneously. Contrast to differential mode.

**common mode input impedance** Common mode input voltage divided by common mode input current.

**common mode rejection ratio** For a differential amplifier, CMRR is the ratio of the common mode gain divided by the differential mode gain. A perfect CMRR would be zero.

**compiler** System software that converts a high-level language program (human readable format) into object code (machine readable format).

**complex instruction set computer (CISC)** A computer with many instructions, instructions that have varying lengths, instructions that execute in varying times, many instructions can access memory, one instruction may both read and write memory, fewer and more specialized registers, and many different types of addressing modes. Contrast to RISC.

**compression ratio** The ratio of the number of original bytes to the number of compressed bytes.

**concurrent programming** A software system that supports two tasks to be active at the same time. A computer with interrupts implements concurrent programming. Compare to distributed and parallel.

**condition code register (CCR)** Register in the processor that contains the status of the previous ALU operation, as well as some operating mode flags such as the interrupt enable bit.

**control coupling** Module A is connected to Module B, because actions in A affect the control path in B.

**control unit (CU)** Component of the processor that determines the sequence of operations.

**cooperative multi-tasking** A scheduler that cannot suspend execution of a thread without the thread's permission. The thread must cooperate and suspend itself. Same as nonpreemptive scheduler.

**counting semaphore** A semaphore that can have any signed integer value. The value  $>0$  means OK and the value  $\leq 0$  means busy. Compare to binary semaphore.

**CPU bound** A situation where the input or output device is faster than the software. In other words, it takes less time for the I/O device to process data, than for the software to process data. Contrast to I/O bound.

**CPU cycle** A memory bus cycle where the address and R/W are controlled by the processor. On microcontrollers without DMA, all cycles are CPU cycles. Contrast to DMA cycle.

**crisp input** An input parameter to the fuzzy logic system, usually with units like cm, cm/sec, °C etc.

**crisp output** An output parameter from the fuzzy logic system, usually with units like dynes, watts etc.

**critical section** Locations within a software module, which if an interrupt were to occur at one of these locations, then an error could occur (e.g., data lost, corrupted data, program crash, etc.) Same as vulnerable window.

**cross-assembler** An assembler that runs on one computer but creates object code for a different computer.

**cross-compiler** A compiler that runs on one computer but creates object code for a different

computer.

**cycle steal DMA** An I/O synchronization scheme that transfers data one item at a time directly from an input device into memory, or directly from memory to an output device. Same as single cycle DMA.

**cycle stretch** The action where some memory cycles are longer allowing time for communication with slower memories, sometimes the memory itself requests the additional time and sometimes the computer has a preprogrammed cycle stretch for certain memory addresses

**DAC** Digital to analog converter, an electronic device that converts digital signals (i.e., integers) to analog form (e.g., voltage).

**data acquisition system** A system that collects information, same as instrument.

**data bus** A set of digital signals that connect the CPU, memory and I/O devices, specifying the value that is being read or written for each bus cycle. See also address bus and control bus.

**data communication equipment (DCE)** A modem or printer connected a serial communication network.

**data terminal equipment (DTE)** A computer or a terminal connected a serial communication network.

**deadline** The time when a task should complete. Compare to slack time.

**dead zone** A condition of a transducer when a large change in the input causes little or no change in the output. Contrast to breakdown.

**deadlock** A scenario that occurs when two or more threads are all blocked each waiting for the other with no hope of recovery.

**defuzzification** Conversion from the fuzzy logic output variables to the crisp outputs.

**desk checking or dry run** We perform a desk check (or dry run) by determining in advance, either by analytical algorithm or explicit calculations, the expected outputs of strategic intermediate stages and final results for a set of typical inputs. We then run our program can compare the actual outputs with this template of expected results.

**device driver** A collection of software routines that perform I/O functions.

**differential mode** For a system with differential inputs, the differential mode properties are defined as signals applied as a difference between the two inputs. Contrast to common mode.

**differential mode input impedance** Differential mode input voltage divided by differential mode input current.

**digital signal processing** Processing of data with digital hardware or software after the signal has been sampled by the ADC, e.g., filters, detection and compression/decompression.

**direct memory access (DMA)** the ability to transfer data between two modules on the bus, this transfer is usually initiated by the hardware (device needs service) and the software configures the communication, but the data is transferred without explicit software action for each piece of data

**direction register** Specifies whether a bi-directional I/O pin is an input or an output. We set a direction register bit to 0 (or 1) to specify the corresponding I/O pin to be input (or output.)

**disarm** Deactivate so that interrupts are not requested, performed by clearing the arm bit.

**Discrete Fourier Transform (DFT)** A technique to convert data in the time domain to data in the frequency domain.  $N$  data points are sampled at  $f_s$ . The resulting frequency resolution is  $f_s / N$ .

**distributed processing** A system implemented across separate computers connected with I/O or a network, so that two or more programs are executed at the same time. Compare to concurrent and parallel.

**DMA** Direct Memory Access is a software/hardware synchronization method where the hardware itself causes a data transfer between the I/O device and memory at the appropriate time when data needs to be transferred. The software usually can perform other work while waiting for the hardware. No software action is required for each individual byte.

**DMA cycle** A memory bus cycle where the address and R/W are controlled by the DMA controller. Contrast to CPU cycle.

**double byte** Two bytes containing 16 bits. Same as halfword.

**double-pole relay** Two separate and complete relays, which are activated together. Contrast to single pole.

**double-pole switch** Two separate and complete switches. The two switches are electrically separate, but mechanically connected. Such that both switches are activated together. Contrast to single pole.

**double-throw relay** A relay with three contact connections, one common and two throws. The common will be connected to exactly one of the two throws (see single-throw).

**double-throw switch** A switch with three contact connections. The center contact will be connected exactly one of the other two contacts. Contrast with single-throw.

**double word** Two words containing 64 bits.

**download** The process of transferring object code from the host (e.g., the PC) to the target microcontroller.

**drop-out** An error that occurs after a right shift or a divide, and the consequence is that an intermediate result loses its ability to represent all of the values. E.g.,  $I=100*(N/51)$  can only result in the values 0, 100, or 200, whereas  $I=(100*N)/51$  properly calculates the desired result.

**dual address DMA** Direct memory access that requires two bus cycles to transfer data from an input device into memory, or from memory to an output device.

**dual port memory** A memory module that interfaces to two separate address/data buses, and allows both systems read/write access the data.

**duty cycle** For a periodic digital wave, it is the percentage of time the signal is high. When an LED display is scanned, it is the percentage of time each LED is active. A motor interfaced using pulse-width-modulation allows the computer to control delivered power by adjusting the duty cycle.

**dynamic allocation** Data structures like the TCB that are created at runtime by calling malloc() and exist until the software releases the memory block back to the heap by calling free(). See static allocation.

**dynamic RAM** Volatile read/write storage built from a capacitor and a single transistor having a low cost, but requiring refresh. Contrast with static RAM.

**EEPROM** Electrically erasable programmable read only memory that is nonvolatile and easy to reprogram. EEPROM can be erased and reprogrammed multiple times. Also see Flash EEPROM.

**embedded computer system** A system that performs a specific dedicated operation where the computer is hidden or embedded inside the machine.

**emulator** An in-circuit emulator is an expensive debugging hardware tool that mimics the processor pin outs. To debug with an emulator, you would remove the processor chip and attach the emulator cable into the processor socket. The emulator would sense the processor input signals and recreate the processor outputs signals on the socket as if an actual processor were actually there, running at full speed. Inside the emulator you have internal read/write access to the registers and processor state. Most emulators allow you to visualize/record strategic information in real-time without halting the program execution. You can also remove ROM chips and insert the connector of a ROM-emulator. This type of emulator is less expensive, and it allows you to debug ROM-based software systems.

**EPROM** Same as PROM. Electrically programmable read only memory that is nonvolatile and requires external devices to erase and reprogram. It is usually erased using UV light.

**erase** The process of clearing the information in a PROM or EEPROM, using electricity or UV light. The information bits are usually all set to logic 1.

**EVB** Evaluation Board, a board-level product used to develop microcontroller systems. Same as LaunchPad.

**even parity** A communication protocol where the number of ones in the data plus a parity bit is an even number. Contrast with odd parity.

**event thread** A thread that is executed or triggered in response to an event. They are similar to ISR, but scheduled by the OS. Typically, the event is a change in hardware status, such as input ready, output idle, or periodically. The trigger could also be a software event. Event threads cannot sleep, block, or be killed. Once they respond to the event, they simply return. Compare to main thread.

**external fragmentation** A condition when the largest file or memory block that can be allocated is less than the total amount of free space on the disk or memory.

**fan out** The number of inputs that a single output can drive if the devices are all in the same logic family.

**Fast Fourier Transform (FFT)** A fast technique to convert data in the time domain to data in the frequency domain.  $N$  data points are sampled at  $f_s$ . The resulting frequency resolution is  $f_s / N$ . Mathematically, the FFT is the same as the DFT, just faster.

**FET** Field effect transistor, also JFET.

**filter** In the debugging context, a filter is a Boolean function or conditional test used to make run-time decisions. For example, if we print information only if two variables  $x, y$  are equal, then the conditional  $(x==y)$  is a filter. Filters can involve hardware status as well.

**finite impulse response filter (FIR)** A digital filter where the output is a function of a finite number of current and past data samples, but not a function of previous filter outputs.

**Finite State Machine (FSM)** An abstract design method to build a machine with inputs and outputs. The machine can be in one of a finite number of states. Which state the system is in represents memory of previous inputs. The output and next state are a function of the input. There may be time delays as well.

**firm real-time** A system that expects all critical tasks to complete on time. Once a deadline has passed, there is no value to completing the task. However, the consequence of missed deadlines is real but the overall system operates with reduced quality. Streaming audio and video are typical examples. Compare to hard real-time and soft real-time.

**fixed-point** A technique where calculations involving nonintegers are performed using a sequence of integer operations. E.g.,  $0.123 * x$  is performed in decimal fixed-point as  $(123 * x) / 1000$  or in binary fixed-point as  $(126 * x) \gg 10$ .

**flash EEPROM** Electrically erasable programmable read only memory that is nonvolatile and easy to reprogram. Flash EEPROMs are typically larger than regular EEPROM.

**floating** A logic state where the output device does not drive high or pull low. The outputs of open collector and tristate devices can be in the floating state. Same as HiZ.

**floor** Establishing a lower bound on the result of an operation. See also ceiling.

**fork** The dynamic action of creating a new thread or process at run time. See also join.

**frame** A complete and distinct packet of bits occurring in a serial communication channel.

**frame** The fixed structure in a relay or transducer. Contrast to armature.

**framing error** An error when the receiver expects a stop bit (1) and the input is 0.

**frequency response** The frequency at which the gain drops to 0.707 of the normal value. For a low pass system, the frequency response ranges from 0 to a maximum value. For a high pass system, the frequency response ranges from a minimum value to infinity. For a bandpass system, the frequency response ranges from a minimum to a maximum value. Same as bandwidth.

**frequency shift key (FSK)** A modem that modulates the digital signals into frequency encoded sine waves.

**friendly** Friendly software modifies just the bits that need to be modified, leaving the other bits unchanged, making it easier to combine modules.

**full duplex channel** Hardware that allows bits (information, error checking, synchronization or overhead) to transfer simultaneously in both directions. Contrast with simplex and half duplex channels.

**full duplex communication** A system that allows information (data, characters) to transfer simultaneously in both directions.

**functional debugging** The process of detecting, locating, or correcting functional and logical errors in a program, typically not involving time. The process of instrumenting a program for such purposes is called functional debugging or often simply debugging.

**fuzzification** Conversion from the crisp inputs to the fuzzy logic input variables.

**fuzzy logic** Boolean logic (true/false) that can take on a range of values from true (255) to false (0). Fuzzy logic **and** is calculated as the minimum. Fuzzy logic **or** is the maximum.

**gadfly** A software/hardware synchronization method where the software continuously reads the hardware status waiting for the hardware operation to complete. The software usually performs no work while waiting for the hardware. Same as busy wait.

**gauge factor** The sensitivity of a strain gauge transducer, i.e., slope of the resistance versus displacement response.

**gibibyte (GiB)**  $2^{30}$  or 1,073,741,824 bytes. Compare to **gigabyte**, which is 1,000,000,000 bytes.

**half duplex channel** Hardware that allows bits (information, error checking, synchronization or overhead) to transfer in both directions, but in only one direction at a



time. Contrast with simplex and full duplex channels.

**half duplex communication** A system that allows information to transfer in both directions, but in only one direction at a time.

**halfword** Two bytes containing 16 bits. Same as double byte.

**handshake** A software/hardware synchronization method where control and status signals go both directions between the transmitter and receiver. The communication is interlocked meaning each device will wait for the other.

**hard real-time** A system that can guarantee that a process will complete a critical task within a certain specified range. In data acquisition systems, hard real-time means there is an upper bound on the latency between when a sample is supposed to be taken (every  $1/f_s$ ) and when the ADC is actually started. Hard real-time also implies that no ADC samples are missed. Compare to hard real-time and firm real-time.

**heartbeat** A debugging monitor, such as a flashing LED, we add for the purpose of seeing if our program is running.

**hexadecimal** A number system that uses base 16.

**HiZ** A logic state where the output device does not drive high or pull low. The outputs of open collector and tristate devices can be in the floating state. Same as floating.

**hold time** When latching data into a device with a rising or falling edge of a clock, the hold time is the time after the active edge of the clock that the data must continue to be valid. See setup time.

**hook** An indirect function call added to a software system that allows the user to attach their programs to run at strategic times. These attachments are created at run time and do not require recompiling the entire system.

**horizontal parity** A parity calculated across the entire message on a bit by bit basis, e.g., the horizontal parity bit 0 is the parity calculated on all the bit 0's of the entire message, can be even or odd parity

**hysteresis** A condition when the output of a system depends not only on the input, but also on the previous outputs, e.g., a transducer that follows a different response curve when the input is increasing than when the input is decreasing.

**I/O bound** A situation where the input or output device is slower than the software. In other words, it takes longer for the I/O device to process data than for the software to process data. Contrast to CPU bound.

**I/O device** Hardware and software components capable of bringing information from the external environment into the computer (input device), or sending data out from the computer to the external environment (output device.)

**I/O port** A hardware device that connects the internal software with external hardware.

$I_{IH}$  Input current when the signal is high.

$I_{IL}$  Input current when the signal is low.

**immediate** An addressing mode where the operand is a fixed data or address value.

**impedance loading** A condition when the input of stage  $n+1$  of an analog system affects the output of stage  $n$ , because the input impedance of stage  $n+1$  is too small and the output impedance of stage  $n$  is too large.

**impedance** The ratio of the effort (voltage, force, pressure) divided by the flow (current, velocity, flow).

**incremental control system** A control system where the actuator has many possible states, and the system increments or decrements the actuator value depending on either in error is positive or negative.

**indexed** An addressing mode where the data or address value for the instruction is located in memory pointed to by an index register.

**infinite impulse response filter (IIR)** A digital filter where the output is a function of an infinite number of past data samples, usually by making the filter output a function of previous filter outputs.

**input bias current** Difference between currents of the two op amp inputs.

**input capture** A mechanism to set a flag and capture the current time (TCNT value) on the rising, falling or rising&falling edge of an external signal. The input capture event can also request an interrupt.

**input impedance** Input voltage divided by input current.

**instruction register (IR)** Register in the control unit that contains the op code for the current instruction.

**instrument** An instrument is the code injected into a program for debugging or profiling. This code is usually extraneous to the normal function of a program and may be temporary or permanent. Instruments injected during interactive sessions are considered to be temporary because these instruments can be removed simply by terminating a session. Instruments injected in source code are considered to be permanent because removal requires editing and recompiling the source. An example of a temporary instrument occurs when the debugger replaces a regular op code with a breakpoint instruction. This temporary instrument can be removed dynamically by restoring the original op code. A print statement added to your source code is an example of a permanent instrument, because removal requires editing and recompiling.

**instrument** An embedded system that collects information, same as data acquisition system.

**instrumentation** The debugging process of injecting or inserting an instrument.

**instrumentation amp** A differential amplifier analog circuit, which can have large gain, large input impedance, small output impedance, and a good common mode rejection ration.

**internal fragmentation** Storage that is allocated for the convenience of the operating system but contains no information. This space is wasted.

**interrupt** A software/hardware synchronization method where the hardware causes a special software program (interrupt handler) to execute when its operation to complete. The software usually can perform other work while waiting for the hardware.

**interrupt flag** A status bit that is set by the timer hardware to signify an external event has occurred.

**interrupt mask** A control bit that, if programmed to 1, will cause an interrupt request when the associated flag is set. Same as **arm**.

**interrupt service routine (ISR)** Program that runs as a result of an interrupt.

**interrupt vector** 32-bit values in ROM specifying where the software should execute after an interrupt request. There is a unique interrupt vector for each type of interrupt including reset.

**intrusive** The debugger itself affects the program being tested. See nonintrusive.

**Inverse Discrete Fourier Transform (IDFT)** A technique to convert data in the frequency

domain to data in the time domain. If there are  $N$  data points and the sampling rate is  $f_s$ , the resulting frequency resolution will be  $f_s / N$ .

**invocation coupling** Module A is connected to Module B, because A calls B.

**I/O mapped I/O** A configuration where the I/O devices are interfaced to the computer in a manner different than the way memories are connected, from an interfacing perspective I/O devices and memory modules have separate bus signals, from a programmer's point of view the I/O devices have their own I/O address map separate from the memory map, and I/O device access requires the use of special I/O instructions

**$I_{OH}$**  Output current when the signal is high. This is the maximum current that has a voltage above  $V_{OH}$ .

**$I_{OL}$**  Output current when the signal is low. This is the maximum current that has a voltage below  $V_{OL}$ .

**jerk** The change in acceleration; the derivative of the acceleration.

**Johnson noise** A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as thermal noise and white noise.

**join** The dynamic action of combining multiple threads or processes at run time. The fork operation will take one thread/process and create multiple children. The join operation will take the multiple threads/processes and convert it back to one. See also fork.

**kibibyte** (KiB)  $2^{10}$  or 1024 bytes. Compare to **kilobyte**, which is 1000 bytes.

**latch** As a noun, it means a register. As a verb, it means to store data into the register.

**latched input port** An input port where the signals are latched (saved) on an edge of an associated strobe signal.

**latency** In this book latency usually refers to the response time of the computer to external events. For example, the time between new input becoming available and the time the input is read by the computer. For example, the time between an output device becoming idle and the time the input is the computer writes new data to it. There can also be latency for an I/O device, which is the response time of the external I/O device hardware to a software command.

**LCD** Liquid Crystal Display, where the computer controls the reflectance or transmittance of the liquid crystal, characterized by its flexible display patterns, low power, and slow speed.

**LED** Light Emitting Diode, where the computer controls the electrical power to the diode, characterized by its simple display patterns, medium power, and high speed.

**light-weight process** Same as a thread.

**linear filter** A filter where the output is a linear combination of its inputs.

**linear recursion** A recursive technique that makes only one call to itself during the execution of the function. Linear recursive functions are easier to implement iteratively. We draw the execution pattern as a straight or linear path. See also recursion, binary recursion, and tail recursion.

**little endian** Mechanism for storing multiple byte numbers such that the least significant byte exists first (in the smallest memory address). Contrast with big endian.

**loader** System software that places the object code into the microcontroller's memory. If the object code is stored in EPROM, the loader is also called a EPROM programmer.

**Local Area Network (LAN)** A connection between computers confined to a small space, such as a room or a building.

**logic analyzer** A hardware debugging tool that allows you to visualize many digital logic signals versus time. Real logic analyzers have at least 32 channels and can have up to 200 channels, with sophisticated techniques for triggering, saving and analyzing the real-time data. In **TEsaS**, logic analyzers have only 8 channels and simply plot digital signals versus time.

**LSB** The least significant bit in a number system is the bit with the smallest significance, usually the right-most bit. With signed or unsigned integers, the significance of the LSB is 1.

**maintenance** Process of verifying, changing, correcting, enhancing, and extending a system.

**make before break** in a double-throw relay or double-throw switch, there is one common contact and two separate contacts. Make before break means as the common contact moves from one of separate contacts to another, it will make (finishing bouncing) the second contact before it breaks off (start bouncing) the first contact. A *form D* relay has a *make before break* operation.

**mailbox** A formal communication structure, similar to a FIFO queue, where the source task puts data into the mailbox and the sink task gets data from the mailbox. The mailbox can hold at most one piece of data at a time, and has two states: mailbox has valid data or mailbox is empty.

**main thread** A thread runs like a main program. If it is static, it is created on startup and runs forever by the scheduler. It could also be dynamic, created at run time. Main threads can sleep, block, and be killed. Compare to event thread.

**mark** A digital value of true or logic 1. Contrast with space.

**mask** As a verb, mask is the operation that selects certain bits out of many bits, using the logical and operation. The bits that are not being selected will be cleared to zero. When used as a noun, mask refers to the specific bits that are being selected.

**Mealy FSM** A FSM where the both the output and next state are a function of the input and state

**measurand** A signal measured by a data acquisition system.

**mebibyte (MiB)**  $2^{20}$  or 1,048,576 bytes. Compare to **megabyte**, which is 1,000,000 bytes.

**membership sets** Fuzzy logic variables that can take on a range of values from true (255) to false (0).

**memory** A computer component capable of storing and recalling information.

**memory-mapped I/O** A configuration where the I/O devices are interfaced to the computer in a manner identical to the way memories are connected, from an interfacing perspective I/O devices and memory modules shares the same bus signals, from a programmer's point of view the I/O devices exist as locations in the memory map, and I/O device access can be performed using any of the memory access instructions.

**microcomputer** A small electronic device capable of performing input/output functions containing a microprocessor, memory, and I/O devices, where small means you can carry it.

**microcontroller** A single chip microcomputer like the TI MSP430, Freescale 9S12, Intel 8051, PIC16, or the Texas Instruments TM4C123.

**mnemonic** The symbolic name of an operation code, like **mov str push**.

**modem** An electronic device that MODulates and DEModulates a communication signal.

Used in serial communication across telephone lines.

**monitor** or **debugger window** A monitor is a debugger feature that allows us to passively view strategic software parameters during the real-time execution of our program. An effective monitor is one that has minimal effect on the performance of the system. When debugging software on a windows-based machine, we can often set up a debugger window that displays the current value of certain software variables.

**Moore FSM** A FSM where the both the output is only a function of the state and the next state is a function of the input and state

**MOSFET** Metal oxide semiconductor field effect transistor.

**MSB** The most significant bit in a number system is the bit with the greatest significance, usually the left-most bit. If the number system is signed, then the MSB signifies positive (0) or negative (1).

**multiple access circular queue** **MACQ** A data structure used in data acquisition systems to hold the current sample and a finite number of previous samples.

**multithreaded** A system with multiple threads (e.g., main program and interrupt service routines) that cooperate towards a common overall goal.

**mutual exclusion** or **mutex** Thread synchronization where at most one thread at a time is allowed to enter.

**negative feedback** An analog system with negative gain feedback paths. These systems are often stable.

**negative logic** A signal where the true value has a lower voltage than the false value, in digital logic true is 0 and false is 1, in TTL logic true is less than 0.7 volts and false is greater than 2 volts, in RS232 protocol true is -12 volts and false is +12 volts. Contrast with positive logic.

**nibble** 4 binary bits or 1 hexadecimal digit.

**nonatomic** Software execution that can be divided or interrupted. Most lines of C code require multiple assembly language instructions to execute, therefore an interrupt may occur in the middle of a line of C code. The instructions store and load multiple, **STM LDM**, are nonatomic.

**nonintrusive** A characteristic when the presence of the collection of information itself does not affect the parameters being measured. Nonintrusiveness is the characteristic or quality of a debugger that allows the software/hardware system to operate normally as if the debugger did not exist. Intrusiveness is used as a measure of the degree of perturbation caused in program performance by an instrument. For example, a print statement added to your source code and single-stepping are very intrusive because they significantly affect the real-time interaction of the hardware and software. When a program interacts with real-time events, the performance is significantly altered. On the other hand, an instrument that toggles an LED on and off (requiring less than a 1  $\mu$ s to execute) is much less intrusive. A logic analyzer that passively monitors the address and data bus is completely nonintrusive. An in-circuit emulator is also non-intrusive because the software input/output relationships will be the same with and without the debugging tool.

**nonlinear filter** A filter where the output is not a linear combination of its inputs. E.g., median, minimum, maximum are examples of nonlinear filters. Contrast to linear filter.

**nonpreemptive scheduler** A scheduler that cannot suspend execution of a thread without

the thread's permission. The thread must cooperate and suspend itself. Same as cooperative multi-tasking.

**nonreentrant** A software module which once started by one thread, should not be interrupted and executed by a second thread. A nonreentrant modules usually involve nonatomic accesses to global variables or I/O ports: read modify write, write followed by read, or a multistep write.

**nonvolatile** A condition where information is not lost when power is removed. When power is restored, then the information is in the state that occurred when the power was removed.

**Nyquist Theorem** If a input signal is captured by an ADC at the regular rate of  $f_s$  samples/sec, then the digital sequence can accurately represent the 0 to  $\frac{1}{2} f_s$  frequency components of the original signal.

**object code** Programs in machine readable format created by the compiler or assembler.

**odd parity** A communication protocol where the number of ones in the data plus a parity bit is an odd number. Contrast with even parity.

**op amp** An integrated analog component with two inputs, ( $V_2, V_1$ ) and an output ( $V_{out}$ ), where  $V_{out} = K \cdot (V_2 - V_1)$ . The amp has a very large gain,  $K$ . Same as operational amplifier.

**op code, opcode, or operation code** A specific instruction executed by the computer. The op code along with the operand completely specifies the function to be performed. In assembly language programming, the op code is represented by its mnemonic, like MOV. During execution, the op code is stored as a machine code loaded in memory.

**open collector** A digital logic output that has two states low and HiZ. Same as open drain and wire-or-mode.

**open drain** A digital logic output that has two states low and HiZ. Same as open collector and wire-or-mode.

**open loop control system** A control system that does not include sensors to measure the current state variables. An analog system with no feedback paths.

**operand** The second part of an instruction that specifies either the data or the address for that instruction. An assembly instruction typically has an op code (e.g., MOV) and an operand (e.g., R0,#55). Instructions that use inherent addressing mode have no operand field.

**operating system** System software for managing computer resources and facilitating common functions like input/output, memory management, and file system.

**originate modem** the device that places the telephone call.

**oscilloscope** A hardware debugging tool that allows you to visualize one or two analog signals versus time.

**output compare** A mechanism to cause a flag to be set and an output pin to change when the timer matches a preset value. The output compare event can also request an interrupt.

**output impedance** Open circuit output voltage divided by short circuit output current.

**overflow** An error that occurs when the result of a calculation exceeds the range of the number system. For example, with 8-bit unsigned integers,  $200+57$  will yield the incorrect result of 1.

**overrun error** An error that occurs when the receiver gets a new frame but the receive FIFO and shift register already have information.

**paged memory** A memory organization where logical addresses (used by software) have multiple and distinct components or fields. The number of bits in the least significant field defines the page size. The physical memory is usually continuous having sequential addresses. There is a dynamic address translation (logical to physical).

**parallel port** A port where all signals are available simultaneously. In this book the ports are 8 bits wide.

**parallel processing** A system that supports two or more programs being executed at the same time. A computer with multiple cores implements parallel programming. Compare to concurrent and distributed.

**partially asynchronous bus** a communication protocol that has a central clock but the memory module can dynamically extend the length of a bus cycle (cycle stretch) if it needs more time

**path expression** A software technique to guarantee subfunctions within a module are executed in a proper sequence. For example, it forces the user to initialize I/O device before attempting to perform I/O.

**PC-relative addressing** An addressing mode where the effective address is calculated by its position relative to the current value of the program counter.

**performance debugging or profiling** The process of acquiring or modifying timing characteristics and execution patterns of a program and the process of instrumenting a program for such purposes is called performance debugging or profiling.

**periodic polling** A software/hardware synchronization method that is a combination of interrupts and busy wait. An interrupt occurs at a regular rate (periodic) independent of the hardware status. The interrupt handler checks the hardware device (polls) to determine if its operation is complete. The software usually can perform other work while waiting for the hardware.

**Personal Area Network (PAN)** A connection between computers controlled by a single person or all working toward for a well-defined single task.

**phase shift key (PSK)** a protocol that encodes the information as phase changes between the sounds.

**photosensor** A transducer that converts reflected or transmitted light into electric current.

**physical plant** The physical device being controlled.

**PID controller** A control system where the actuator output depends on a linear combination of the current error (P), the integral of the error (I) and the derivative of the error (D).

**pink noise** A fundamental noise in resistive devices arising from fluctuating conductivity. Same as 1/f noise.

**pole** A place in the frequency domain where the filter gain is infinite.

**polling** A software function to look and see which of the potential sources requested the interrupt.

**port** External pins through which the microcontroller can perform input/output. Same as I/O port.

**positive feedback** An analog system with positive gain feedback paths. These systems will saturate.

**positive logic** a signal where the true value has a higher voltage than the false value, in digital logic true is 1 and false is 0, in TTL logic true is greater than 2 volts and false is

less than 0.7 volts, in RS232 protocol true is +12 volts and false is -12 volts. Contrast with negative logic.

**potentiometer** A transducer that converts position into electric resistance.

**precision** A term specifying the degrees of freedom from random errors. For an input signal, it is the number of distinguishable input signals that can be reliably detected by the measurement. For an output signal, it is the number of different output parameters that can be produced by the system. For a number system, precision is the number of distinct or different values of a number system in units of “alternatives”. The precision of a number system is also the number of binary digits required to represent all its numbers in units of “bits”.

**preemptive scheduler** A scheduler that has the power to suspend execution of a thread without the thread's permission.

**priority** When two requests for service are made simultaneously, priority determines which order to process them.

**private** Can be accessed only by software modules in that local group.

**private variable** A global variable that is used by a single thread, and not shared with other threads.

**process** The execution of software that does not necessarily cooperate with other processes.

**producer-consumer** A multithreaded system where the producers generate new data, and the consumers process or output the data.

**profile** A collection of services implemented by Bluetooth.

**profiling** See performance debugging.

**program counter (PC)** A register in the processor that points to the memory containing the instruction to execute next.

**PROM** Same as EPROM. Programmable read only memory that is nonvolatile and requires external devices to erase and reprogram. It is usually erased using UV light.

**promotion** Increasing the precision of a number for convenience or to avoid overflow errors during calculations.

**pseudo interrupt vector** A secondary place for the interrupt vectors for the convenience of the debugger, because the debugger cannot or does not want the user to modify the real interrupt vectors. They provide flexibility for debugging but incur a run time delay during execution.

**pseudo op** Operations included in the program that are not executed by the computer at run time, but rather are interpreted by the assembler during the assembly process. Same as assembly directive.

**pseudo-code** A shorthand for describing a software algorithm. The exact format is not defined, but many programmers use their favorite high-level language syntax (like C) without paying rigorous attention to the punctuation.

**public** Can be accessed by any software module.

**public variable** A global variable that is shared by multiple programs or threads.

**pulse width modulation** A technique to deliver a variable signal (voltage, power, energy) using an on/off signal with a variable percentage of time the signal is on (duty cycle). Same as **variable duty cycle**.



**Q** The Q of a bandpass filter (passes  $f_{\min}$  to  $f_{\max}$ ) is the center pass frequency ( $f_o = (f_{\max} + f_{\min})/2$ ) divided by the width of the pass region,  $Q = f_o / (f_{\max} - f_{\min})$ . The Q of a bandreject filter (rejects  $f_{\min}$  to  $f_{\max}$ ) is the center reject frequency ( $f_o = (f_{\max} + f_{\min})/2$ ) divided by the width of the reject region,  $Q = f_o / (f_{\max} - f_{\min})$ .

**quadrature amplitude modem (QAM)** a protocol that used both the phase and amplitude to encode up to 6 bits onto each baud.

**qualitative DAS** A DAS that collects information not in the form of numerical values, but rather in the form of the qualitative senses, e.g., sight, hearing, smell, taste and touch. A qualitative DAS may also detect the presence or absence of conditions.

**quantitative DAS** A DAS that collects information in the form of numerical values.

**RAM** Random Access Memory, a type of memory where the information can be stored and retrieved easily and quickly. Since it is volatile the information is lost when power is removed.

**range** Includes both the smallest possible and the largest possible signal (input or output). The difference between the largest and smallest input that can be measured by the instrument. The units are in the units of the measurand. When precision is in alternatives,  $\text{range} = \text{precision} \cdot \text{resolution}$ . Same as span

**read cycle** data flows from the memory or input device to the processor, the address bus specifies the memory or input device location and the data bus contains the information at that address

**read data available** The time interval (start,end) during which the data will be valid during a read cycle, determined by the memory module

**real-time** A characteristic of a system that can guarantee an upper bound (worst case) on latency.

**real-time system** A system where time-critical operations occur when needed.

**recursion** A programming technique where a function calls itself.

**reduced instruction set computer (RISC)** A computer with a few instructions, instructions with fixed lengths, instructions that execute in 1 or 2 bus cycles, only load and store can access memory, no one instruction can both read and write memory, many identical general purpose registers, and a limited number of addressing modes. Contrast to CISC.

**reentrant** A software module that can be started by one thread, interrupted and executed by a second thread. A reentrant module allows both threads to properly execute the desired function. Contrast with non-reentrant.

**registers** High-speed memory located in the processor. The registers in the ARM® Cortex™-M include R0 through R15.

**relay** A mechanical switch that can be turned on and off by the computer.

**reliability** The ability of a system to operate within specified parameters for a stated period of time. Given in terms of mean time between failures (MTBF).

**reproducibility (or repeatability)** A parameter specifying how consistent over time the measurement is when the input remains fixed.

**requirements document** A formal description of what the system will do in a very complete way, but not including how it will be done. It should be unambiguous, complete, verifiable, and modifiable.

**reset vector** The 32-bit value at memory locations 0x0000.0004 specifying where the

software should start after power is turned on or after a hardware reset.

**resolution** For an input signal, it is the smallest change in the input parameter that can be reliably detected by the measurement. For an output signal, it is the smallest change in the output parameter that can be produced by the system, range equals precision times resolution. The units are in the units of the measurand. When precision is in alternatives,  $\text{range} = \text{precision} \cdot \text{resolution}$ .

**response time** Similar to latency, it is the delay between when the time an event occurs and the time the software responds to the event.

**ritual** Software, usually executed once at the beginning of the program, that defines the operational modes of the I/O ports.

**ROM** Read Only Memory, a type of memory where the information is programmed into the device once, but can be accessed quickly. It is low cost, must be purchased in high volume and can be programmed only once. See also EPROM, EEPROM, and flash EEPROM.

**rotor** The part of a motor that rotates.

**round robin scheduler** A scheduler that runs each active thread equally.

**roundoff** The error that occurs in a fixed-point or floating-point calculation when the least significant bits of an intermediate calculation are discarded so the result can fit into the finite precision.

**sample and hold** A circuit used to latch a rapidly changing analog signal, capturing its input value and holding its output constant.

**sampling rate** The rate at which data is collected in a data acquisition system.

**saturation** A device that is no longer sensitive to its inputs when its input goes above a maximum value or below a minimum value.

**scan or scanpoint** Any instrument used to produce a side effect without causing a break (halt) is a scan. Therefore, a scan may be used to gather data passively or to modify functions of a program. Examples include software added to your source code that simply outputs or modifies a global variable without halting. A scanpoint is triggered in a manner similar to a breakpoint but a scanpoint simply records data at that time without halting execution.

**scheduler** System software that suspends and launches threads.

**Schmitt Trigger** A digital interface with hysteresis making it less susceptible to noise.

**scope** A logic analyzer or an oscilloscope, hardware debugging tools that allows you to visualize multiple digital or analog signals versus time.

**select signal** The output of the address decoder (each module on the bus has a separate address decoder); a Boolean (true/false) signal specifying whether or not the current address of the bus matches the device address

**semaphore** A system function with two operations (wait and signal) that provide for thread synchronization and resource sharing.

**sensitivity** The sensitivity of a transducer is the slope of the output versus input response. The sensitivity of a qualitative DAS that detects events is the percentage of actual events that are properly recognized by the system.

**serial communication** A process where information is transmitted one bit at a time.

**serial peripheral interface (SPI)** A device to transmit data with synchronous serial

communication protocol. Same as SSI.

**serial port** An I/O port with which the bits are input or output one at a time.

**service** A collection of Bluetooth functionalities that taken together solve one coherent system function. Examples include blood pressure monitor and human interface device (mouse, keyboard).

**servo** A DC motor with built in controller. The microcontroller specifies desired position and the servo adds/subtracts power to move the shaft to that position.

**setup time** When latching data into a register with a clock, it is the time before an edge the input must be valid. Contrast with hold time.

**shot noise** A fundamental noise that occurs in devices that count discrete events.

**signed two's complement binary** A mechanism to represent signed integers where 1 followed by all 0's is the most negative number, all 1's represents the value -1, all 0's represents the value 0, and 0 followed by all 1's is the largest positive number.

**sign-magnitude binary** A mechanism to represent signed integers where the most significant bit is set if the number is negative, and the remaining bits represent the magnitude as an unsigned binary.

**simplex channel** Hardware that allows bits (information, error checking, synchronization or overhead) to transfer only in one direction. Contrast with half duplex and full duplex channels.

**simplex communication** A system that allows information to transfer only in one direction.

**simulator** A simulator is a software application that simulates or mimics the operation of a processor or computer system. Most simulators recreate only simple I/O ports and often do not effectively duplicate the real-time interactions of the software/hardware interface. On the other hand, they do provide a simple and interactive mechanism to test software. Simulators are especially useful when learning a new language, because they provide more control and access to the simulated machine, than one normally has with real hardware.

**single address DMA** Direct memory access that requires only one bus cycle to transfer data from an input device into memory, or from memory to an output device.

**single cycle DMA** An I/O synchronization scheme that transfers data one item at a time directly from an input device into memory, or directly from memory to an output device. Same as cycle steal DMA.

**single-pole relay** A simple relay with only one copy of the switch mechanism. Contrast with double pole.

**single-pole switch** A simple switch with only one copy of the switch mechanism. One switch that acts independent from other switches in the system. Contrast with double-pole.

**single-throw switch** A switch with two contact connections. The two contacts may be connected or disconnected. Contrast with double-throw.

**slack time** The time-to-deadline minus the how long it will take to complete the task. For example, if slack time is zero, then you could complete the task on time if you devoted 100% of your resources.

**slew rate** The maximum slope of a signal. If the time-varying signal  $V(t)$  is in volts, the slew rate is the maximum  $dV/dt$  in volts/s.

**soft real-time** A system that implements best effort to execute critical tasks on time, typically using a priority scheduler. Once a deadline as passed, the value of completing the

task diminishes over time. Compare to hard real-time and firm real-time.

**software interrupt** A software interrupt is similar to a regular or hardware interrupt: there is a trigger that invokes the execution of an ISR. On the Cortex™-M, there are two software interrupts: supervisor call and PendSV (vectors at 0x00000028 and 0x00000038 respectively). The difference between hardware and software interrupts is the trigger. Hardware interrupts are triggered by hardware events, while software interrupts are triggered explicitly by software. For example, to invoke a PendSV, the software sets bit 28 of the NVIC\_INT\_CTRL\_R register. Same as trap.

**software maintenance** Process of verifying, changing, correcting, enhancing, and extending software.

**solenoid** A discrete motion device (on/off) that can be controlled by the computer usually by activating an electromagnet. For example, electronic door locks on automobiles.

**source code** Programs in human readable format created with an editor.

**space** A digital value of false or logic 0. Contrast with mark.

**span** Same as range.

**spatial resolution** The volume over which the DAS collects information about the measurand.

**specificity** The specificity of a transducer is the relative sensitivity of the device to the signal of interest versus the sensitivity of the device to other unwanted signals. The sensitivity of a qualitative DAS that detects events is the percentage of events detected by the system that are actually true events.

**spinlock semaphore** A semaphore where the threads will spin (run but do no useful function) when they execute wait on a busy semaphore. Contrast to blocking semaphore.

**stabilize** The debugging process of stabilizing a software system involves specifying all its inputs. When a system is stabilized, the output results are consistently repeatable. Stabilizing a system with multiple real-time events, like input devices and time-dependent conditions, can be difficult to accomplish. It often involves replacing input hardware with sequential reads from an array or disk file.

**stack** Last in first out data structure located in RAM and used to temporarily save information.

**stack pointer (SP)** A register in the processor that points to the RAM location of the stack.

**start bit** An overhead bit(s) specifying the beginning of the frame, used in serial communication to synchronize the receiver shift register with the transmitter clock. See also stop bit, even parity and odd parity.

**starvation** A condition that occurs with a priority scheduler where low priority threads are never run.

**static allocation** Data structures such as an FSM or TCB that are defined at assembly or compile time and exist throughout the life of the software. Contrast to dynamic allocation.

**static RAM** Volatile read/write storage built from three transistors having fast speed, and not requiring refresh. Contrast with dynamic RAM.

**stator** The part of a motor that remains stationary. Same as frame.

**stepper motor** A motor that moves in discrete steps.

**stop bit** An overhead bit(s) specifying the end of the frame, used in serial communication to separate one frame from the next. See also start bit, even parity and odd parity.

**strain gauge** A transducer that converts displacement into electric resistance. It can also be used to measure force or pressure.

**string** A sequence of ASCII characters, usually terminated with a zero.

**symbol table** A mapping from a symbolic name to its corresponding 16-bit address, generated by the assembler in pass one and displayed in the listing file.

**synchronous bus** a communication protocol that has a central clock; there is no feedback from the memory to the processor, so every memory cycle takes exactly the same time; data transfers (put data on bus, take data off bus) are synchronized to the central clock

**synchronous protocol** a system where the two devices share the same clock.

**synchronous serial interface (SSI)** A device to transmit data with synchronous serial communication protocol. Same as SPI.

**tachometer** a sensor that measures the revolutions per second of a rotating shaft.

**tail recursion** A technique where the recursive call occurs as the last action taken by the function. See also recursion, binary recursion, and linear recursion.

**thermal noise** A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as Johnson noise and white noise.

**thermistor** A nonlinear transducer that converts temperature into electric resistance.

**thermocouple** A transducer that converts temperature into electric voltage.

**thread** The execution of software that cooperates with other threads. A thread embodies the action of the software. One concept describes a thread as the sequence of operations including the input and output data.

**thread control block TCB** Information about each thread.

**three-pole relay** Three separate and complete relays, which are activated together (see single pole).

**three-pole switch** Three separate and complete switches. The switches are electrically separate, but mechanically connected. The three switches turned on and off together (see single pole).

**throughput** The information transfer rate, the amount of data transferred per second. Same as bandwidth.

**time constant** The time to reach 63.2% of the final output after the input is instantaneously increased.

**time profile and execution profile** Time profile refers to the timing characteristic of a program and execution profile refers to the execution pattern of a program.

**time to deadline** The time between now and the deadline.

**tolerance** The maximum deviation of a parameter from a specified value.

**total harmonic distortion (THD)** A measure of the harmonic distortion present and is defined as the ratio of the sum of the powers of all harmonic components to the power of the fundamental frequency.

**transducer** A device that converts one type of signal into another type.

**trap** A trap is similar to a regular or hardware interrupt: there is a trigger that invokes the execution of an ISR. On the Cortex<sup>TM</sup>-M, there are two software interrupts: supervisor call and PendSV (vectors at 0x00000028 and 0x00000038 respectively). The difference between hardware and software interrupts is the trigger. Hardware interrupts are triggered by hardware events, while software interrupts are triggered explicitly by software. For

example, to invoke a PendSV, the software sets bit 28 of the NVIC\_INT\_CTRL\_R register. Same as software interrupt.

**tristate** The state of a tristate logic output when off or not driven.

**tristate logic** A digital logic device that has three output states low, high, and off (HiZ).

**truncation** The act of discarding bits as a number is converted from one format to another.

**two's complement** A number system used to define signed integers. The MSB defines whether the number is negative (1) or positive (0). To negate a two's complement number, one first complements (flip from 0 to 1 or from 1 to 0) each bit, then add 1 to the number.

**two-pole relay** two separate and complete relays, which are activated together (same as double pole).

**two-pole switch** Two separate and complete switches. The switches are electrically separate, but mechanically connected. The two switches turned on and off together which are activated together, same as double-pole.

**ultrasound** A sound with a frequency too high to be heard by humans, typically 40 kHz to 100 MHz.

**unbuffered I/O** The hardware and software are tightly coupled so that both wait for each other during the transmission of data.

**unipolar stepper motor** A stepper motor where the current flows in only one direction (on/off) along the interface wires; a stepper with 5 or 6 interface wires.

**universal asynchronous receiver/transmitter (UART)** A device to transmit data with asynchronous serial communication protocol, same as ACIA.

**unsigned binary** A mechanism to represent unsigned integers where all 0's represents the value 0, and all 1's represents is the largest positive number.

**vector** A 32-bit address in ROM containing the location of the interrupt service routines. See also reset vector and interrupt vector.

**velocity factor (VF)** The ratio of the speed at which information travels relative to the speed of light.

**vertical parity** The normal parity bit calculated on each individual frame, can be even or odd parity

$V_{OH}$  The smallest possible output voltage when the signal is high, and the current is less than  $I_{OH}$ .

$V_{OL}$  The largest possible output voltage when the signal is low, and the current is less than  $I_{OL}$ .

**volatile** A condition where information is lost when power is removed.

**volatile** A property of a variable in C, such that the value of the variable can change outside the immediate scope of the software accessing the variable.

**voltage follower** An analog circuit with gain equal to 1, large input impedance and small output impedance. Same as follower.

**vulnerable window** Locations within a software module, which if an interrupt were to occur at one of these locations, then an error could occur (e.g., data lost, corrupted data, program crash, etc.) Same as critical section.

**white noise** A fundamental noise in resistive devices arising from the uncertainty about the position and velocity of individual molecules. Same as Johnson noise and thermal noise.

**wire-or-mode** A digital logic output that has two states low and HiZ. Same as open collector.

**word** Four bytes containing 32 bits.

**workstation** A powerful general purpose computer system having a price in the \$3K to 50K range and used for handling large amounts of data and performing many calculations.

**write cycle** data is sent from the processor to the memory or output device, the address bus specifies the memory or output device location and the data bus contains the information

**write data available** time interval (start,end) during which the data will be valid during a write cycle, determined by the processor

**write data required** time interval (start,end) during which the data should be valid during a write cycle, determined by the memory module

**XON/XOFF** A protocol used by printers to feedback the printer status to the computer. XOFF is sent from the printer to the computer in order to stop data transfer, and XON is sent from the printer to the computer in order to resume data transfer.

**Z Transform** A transform equation converting a digital time-domain sequence into the frequency domain. In both the time and frequency domain it is assumed the signal is band limited to 0 to  $\frac{1}{2}f_s$ .

**zero** A place in the frequency domain where the filter gain is zero.

## Appendix 2. Solutions to Checkpoints

**Checkpoint 1.1:** A characteristic of a system that can guarantee that important tasks get run at the correct time. We define latency as the difference between the time a task is scheduled to run, and the time when the task is actually run. A real-time system guarantees the latency will be small and bounded.

**Checkpoint 1.2:** An embedded system performs a specific dedicated operation where the computer is hidden or embedded inside the machine.

**Checkpoint 1.3:** Minimize size, minimize weight, minimize power, provide for proper operation in harsh environments, maximize safety, and minimize cost.

**Checkpoint 1.4:** Multiple busses allow multiple operations to occur in parallel, resulting in higher performance (more operations/sec).

**Checkpoint 1.5:** The system does not run slower during debugging, because debugger functions occur simultaneously with program operation.

**Checkpoint 1.6:** Variables, the heap, and the stack go in RAM. Constants and machine code go in ROM. Basically, items that can change over time go in RAM and items that do not change go in ROM.

**Checkpoint 1.7:** The ROM on our microcontroller is electrically erasable programmable read only memory (EEPROM). So yes the software can erase the memory and reprogram it. Under normal conditions however software does not write to ROM. However, you can create a file system using a piece of ROM, where your software will be writing to ROM..

**Checkpoint 1.8:**  $0x2200.0000 + 32*n + 4*b = 0x2200.0000 + 32*0x1010 + 4*3 = 0x2200.0000 + 0x20200 + 0x0C = 0x2202.020C$ .

**Checkpoint 1.9:**  $0x2200.0000 + 32*n + 4*b = 0x2200.0000 + 32*0x10000 + 4*22 = 0x2200.0000 + 0x200000 + 0x58 = 0x2220.0058$ .

**Checkpoint 1.10:**  $0x4200.0000 + 32*n + 4*b = 0x4200.0000 + 32*0x30 + 4*7 = 0x4200.0000 + 0x00600 + 0x1C = 0x4200.061C$ .

**Checkpoint 1.11:** R13 is the stack pointer, used to create temporary storage (also called SP). R14 is the link register (also called LR), containing the return address when a function is called. R15 is the program counter, containing the address of the instruction as software executes (also called PC).

**Checkpoint 1.12:** The I bit in bit 0 of the PRIMASK register. If I=0 interrupts are enabled. If I=1 interrupts are disabled (postponed).

**Checkpoint 1.13:** A pin is an individual wire on the microcontroller, pins can be used for input, output, debugging, or power. A port is a collection of input/output pins with a common operation.

**Checkpoint 1.14:** Parallel, serial, analog and time.

**Checkpoint 1.15:** The addressing mode specifies how the instruction accesses data.

**Checkpoint 1.16:** Data are numbers and addresses are memory locations that point to data. The processor does not know if a value in R0 is data or an address. It is the responsibility of the programmer (you) to use data as numbers and addresses as pointers in the way you write your programs.

**Checkpoint 1.17:** Since this instruction pushes 4 registers, the SP is decremented by 16.

**Checkpoint 1.18:** The return address is saved in the link register, R14 or LR. However, when a first function calls a second function, the first function must save the LR onto the stack.



**Checkpoint 1.19:** Standards allows software written by one company to work properly with software written by another company. A similar concept is CMSIS, which allows the standardization of I/O functions, see <http://www.keil.com/pack/doc/CMSIS/General/html/index.html>.

**Checkpoint 1.20:** A pointer is an address that points to data. Pointers are important because they allow us to pass large amounts of data with a single 32-bit entity.

**Checkpoint 1.21:** An array of 10 elements is accessed with indices from 0 to 9.

**Checkpoint 1.22:** A linked list is a collection of nodes, where each node contains data and a pointer to the next node. The advantage of linked list is the data can grow and shrink in size, and you can sort the order dynamically. In real-time systems we must guarantee execution of important tasks occur at the proper time, so we will be careful when implementing flexible behavior, which in some instances may not finish. Sometimes we sacrifice flexibility of linked lists for the stability and simplicity of arrays.

**Checkpoint 1.23:** This is internal fragmentation because it is wasted space for efficiency or the convenience of the operating system.

**Checkpoint 1.24:** Search the free list to see if the address **&Heap[SIZE\*i]** is free.

**Checkpoint 1.25:** Ignore size parameter, return 100 bytes regardless of the request.

**Checkpoint 1.26:** The block is lost. This is an example of a memory leak.

**Checkpoint 1.27:** Sort the free blocks by size using a binary tree. This way it will be faster to search for the best free block during allocation.

**Checkpoint 1.28:** Each block has two counters. Dividing a block into two creates one more block. There needs to be two more counters for the new block.

**Checkpoint 1.29:** Weird and crazy bugs will occur, because that memory may be allocated to another task.

**Checkpoint 1.30:** The existence of the instrument has a small but inconsequential effect on the system performance. The time to execute the instrument is small compared to the time between executions of the instrument. There are three advantages of leaving the instruments in the final system. First, the system was tested with the instruments and works to specification with the instruments. There is no guarantee the system will still work if the instruments are removed. Second, the instruments could provide run time checks to catch failures during operation. Third, the instruments could be used during system checkup (recalibration, diagnostic checkup etc.)

**Checkpoint 2.1:** Not all pins of a port must have the same direction. Some may be inputs while others are outputs. Furthermore, some pins may be off, meaning neither input or output.

**Checkpoint 2.2:** If we activate the HFXT to run the microcontroller at 48 MHz, then the SysTick counter decrements every 20.83 ns. To make it interrupt every 10ms, it should interrupt every 480000 cycles. Thus, we set reload to 479999.

**Checkpoint 2.3:** Since real-time events trigger interrupts, and the ISR software services the requests, disabling interrupts will postpone the response causing latency or jitter. The maximum jitter will be the maximum time running with interrupts disabled.

**Checkpoint 2.4:** Notice there are two disable interrupt and two enable interrupt functions, occurring in this order: 1) disable, 2) disable, 3) enable, 4) enable. Interrupts will be incorrectly enabled after step 3). Since the 1-4 represents a critical section and 2-3 is inside this section, a bug will probably be introduced. In this example **Stuff1B** runs with interrupts enabled.

<b>Critical1</b>	<b>Critical2</b>
<b>Disable // 1</b>	<b>Disable // 2</b>
<b>Stuff1A</b>	<b>Stuff2A</b>
<b>Call Critical2</b>	<b>Enable // 3</b>
<b>Stuff1B</b>	<b>return</b>
<b>Enable // 4</b>	
<b>return</b>	

**Checkpoint 2.5:** Negative logic means when we touch the switch the voltage goes to 0 (low). Formally, negative logic means the true voltage is lower than the false voltage. Positive logic means when we touch the switch the voltage goes to +3.3 (high). Formally, positive logic means the true voltage is higher than the false voltage.

**Checkpoint 2.6:** For PF4, we need input with pull-up. DIR bit 4 is low (input), AFSEL bit 4 is low (not alternate), PUE bit 4 high (pull-up) and PDE bit 4 low (not pull-down). For PF0, we also need input with pull-down. DIR bit 0 is low (input), AFSEL bit 0 is low (not alternate), PUE bit 0 high (pull-up) and PDE bit 0 low (not pull-down).

**Checkpoint 2.7:** For the TM4C one interrupt is generated, both flags are set, and both counts will be increments. Compare this to the MSP432 version that will generate two sequential interrupts and each interrupt will service one request. In both cases, no events are lost.

**Checkpoint 2.8:** There is 1 byte of data per 10 bits of transmission. So, there are 11520 bytes/sec.

**Checkpoint 2.9:** The Rx Fifo is empty when there is no input data. Software is waiting for hardware. We classify this condition as I/O bound, because the system bandwidth is limited by I/O hardware.

**Checkpoint 2.10:** The Tx Fifo is empty when there is no output data. Hardware is waiting for software. We classify this condition as CPU bound, because the system bandwidth is limited by software execution speed.

**Checkpoint 2.11:** PWM: on the cycle when the timer equals the value in the Match Register or the Interval Load Register.

**Checkpoint 2.12:** PWM: output pin cleared (set if inverting mode) on match or set (cleared if inverting mode) on reload.

**Checkpoint 2.13:**  $1V \cdot 16384 / 2.5V = 6553$  (or 6554). The TM4C range is 0 to 3.3V,  $1V \cdot 4095 / 3.3V = 1241$ .

**Checkpoint 2.14:** `P1OUT ^= 0x08; GPIO_PORTA_DATA_R ^= 0x08;`

`#define PA3 (*(volatile uint32_t *)0x40000020)`

`#define Debug_HeartBeat() (PA3 ^= 0x08)`

**Checkpoint 3.1:** A program is a list of commands, while a thread is the action cause by the execution of software. For example, there might be one copy of a program that searches the card catalog of a library, while separate threads are created for each user that logs into a terminal to perform a search. Similarly, there might be one set of programs that implement the features of a window (open, minimize, maximize, etc.), while there will be a separate thread for each window created.

**Checkpoint 3.2:** Threads can't communicate with each other using the stack, because they have physically separate stacks. Global variables will be used, because one thread may write to the

global, and another can read from it.

**Checkpoint 3.3:** It is hard real time because if the response is late, data may be lost.

**Checkpoint 3.4:** It is firm real time because it causes an error that can be perceived but the effect is harmless and does not significantly alter the quality of the experience.

**Checkpoint 3.5:** It is soft real time because the faster it responds the better, but the value of the system (bandwidth is amount of data printed per second) diminishes with latency.

**Checkpoint 3.6:** With the flowchart in Figure 3.8, the Status will be set twice and the first data value will be lost. We will fix this error in the next using a first in first out (FIFO) queue.

**Checkpoint 3.7:** The system will not work, because there is more work to do than there are processor resources to accomplish them.

**Checkpoint 3.8:** The system will work some of the time, but there are times the system will not work.

**Checkpoint 3.9:** The function **OS\_Wait** will crash because it is spinning with interrupts disabled.

**Checkpoint 3.10:** The function **OS\_Wait** has a critical section around the read-modify-write access to the semaphore. If we remove the mutual exclusion, multiple threads could pass.

**Checkpoint 3.11:** Notice this function discards the new data on error

```
void SendMail(uint32_t int data){
    if(Send){
        Lost++; // discard new data
    }else{
        Mail = data;
        OS_Signal(&Send);
    }
}
```

**Checkpoint 4.1:** Each thread runs for 1ms, so each thread runs every 5ms. The spinning thread will be run 200 times, wasting 200ms while it waits for its semaphore to be signaled. This is a 20% waste of processor time.

**Checkpoint 4.2:** Other threads run for 1 ms each, the semaphore is checked every 4 ms. However, the amount of time wasted will be quite small because the spinning thread will go through the loop once and suspend. Obviously, once the semaphore goes above 0, the **OS\_Wait** will return.

**Checkpoint 4.3:** The worst case is you must look at all 5 blocked threads, so the while loop executes 5 times. This is a waste of  $5 \times 150 = 750$ ns. Since the scheduler runs every 1 ms, this waste is 0.075% of processor time.

**Checkpoint 4.4:** Since Signal increments and Wait decrements, we expect the average to be equal. On average, over a long period of time, the number of calls to Wait equals the number of calls to Signal. If Signal were called more often, then the semaphore value would become infinite. If Wait were called more often, then all threads would become blocked/stalled.

**Checkpoint 4.5:** Since put enters data and get removes, we expect the average to be equal. If put were called more often, then the FIFO would become full and another call to put could not occur. If get were called more often, then FIFO would become empty and another successful call to get

could not occur. If the FIFO can store  $N$  pieces of data, then the total number of successful puts minus the total number of successful gets must be a value between 0 and  $N$ . On average, over a long period of time, the number of calls to put equals the number of calls to get.

**Checkpoint 4.6:** If CurrentSize is 0, the FIFO is empty. If CurrentSize is equal to FIFOSIZE, the FIFO is full.

**Checkpoint 4.7:** Use AND instead of modulo divide when incrementing the index because it is faster

**PutI = (PutI+1)&(FIFOSIZE-1);**

**GetI = (GetI+1)&(FIFOSIZE-1);**

**Checkpoint 5.1:** This priority scheduler must look at them all, so it will run  $N$  times through the loop. Looking at all the threads is ok if  $N$  is small, but becomes inefficient if  $I$  is large.

**Checkpoint 5.2:** The maximum latency is 20 ms, because the switch will be recognized at the next interrupt. The minimum latency is 0, and the latencies are uniformly distributed from 0 to 20, so the average is 10 ms.

**Checkpoint 6.1:** At 60 Hz,  $f/f_s$  is  $1/6$ .

$$\text{Gain} = 0.5 \sqrt{\{1 + \cos(6\pi/6)\}^2 + \{\sin(6\pi/6)\}^2} = \sqrt{\{1-1\}^2 + \{0\}^2} = 0$$

**Checkpoint 6.2:** If the gain is larger than one, amplification occurs. For example, if the gain is 1.2, if you put in a sinusoidal wave with amplitude 100, then the output of the filter will be a sinusoidal wave with amplitude 120. This is important because a filtered signal from an 8-bit ADC will not fit into an 8-bit variable.

**Checkpoint 6.3:** The  $Q$  is much higher for the IIR filter. This means it rejects just 60 Hz, and passes most of the other frequencies. This greatly improved performance comes with only a modest increase the computational complexity. The additional computation is 2 multiplies and a subtraction. The performance for the IIR filter is superior.

**Checkpoint 6.4:** First, sum all the positive terms, 76050. The largest positive value will be if the ADC values for the positive terms are 4095 and the ADC values for the negative terms is 0.  $76050 \cdot 4095$  is less than  $2^{31}$ . Next, sum all the negative terms, -76048. The largest negative value will be if the ADC values for the negative terms are 4095 and the ADC values for the positive terms is 0.  $-76048 \cdot 4095$  is greater than  $-2^{31}$ . The input is bounded from 0 to 4095 because the data comes from the 12-bit ADC. The largest gain in this filter is 5, the fixed-point coefficient is 16384.  $4095 \cdot 5 \cdot 16384$  will fit in the 32-bit signed intermediate result, **sum**.

**Checkpoint 6.5:** Because of the linear phase the  $h(n)$  filter coefficients are symmetric. Notice that  $h(k)$  equals  $h(50-k)$ . For example,  $4 \cdot x(n) + 4 \cdot x(n-50)$  can be replaced with  $4 \cdot (x(n) + x(n-50))$ . In general,  $h(k) \cdot x(n-k) + h(50-k) \cdot x(n-50-k)$  can be replaced with  $h(k) \cdot (x(n-k) + x(n-50-k))$ , saving 25 multiplies.

**Checkpoint 7.1:** Both refer to the speed of communication. Latency is the response time to a question and bandwidth is the information transfer rate.

**Checkpoint 7.2:** If we do not meet the latency requirement, that data is lost. If it happens every time the system doesn't work. If it happens occasionally, it will run slow because we will have to wait for the disk to spin around one revolution and try it again.

**Checkpoint 7.3:** A portion of the sound is lost, and it will sound like a skip. We may also hear a

click because the waveform is discontinuous. It is firm real time because it causes an error that can be perceived but the effect is harmless and does not significantly alter the quality of the experience.

**Checkpoint 7.4:** The system runs slow, because the transmitter will timeout and try to resend the packets.

**Checkpoint 7.5:** The bidirectional driver has three possibilities, determined by two control pins. An example of this type of logic is the 74HC245. It can drive data left to right, making the left input and right output. It can drive data right to left, making the right input and left output. The third possibility is that the device can be off, driving neither the left nor the right. This is a noninverting driver, so the output equals the input.

**Checkpoint 7.6:** Substitute the four bidirectional data bus drivers with four unidirectional tristate drivers. All four data bus drivers operate in the direction of the simplex transfer (left to right). The bank-switched memory looks like a write-only memory to the computer and a read-only memory to the I/O hardware.

**Checkpoint 7.7:** The maximum latency for cycle steal DMA is one bus cycle, assume there is only one DMA channel active. If there is more than one DMA channel operating, one DMA request may have to wait for another.

**Checkpoint 7.8:** On some systems the latency is only one bus cycle. On others it may be 2 or 3 bus cycles. In all cases it is very short.

**Checkpoint 7.9:** On most systems, the instruction must finish, so the latency will be the maximum instruction length. In all cases burst DMA has a longer latency than cycle steal.

**Checkpoint 8.1:** On average, each file wastes  $\frac{1}{2}n$  bytes. Since this is inside the file, this wasted space is classified as internal fragmentation.

**Checkpoint 8.2:** The best way to cut the wood is obviously at one end or the other, generating the 2-meter piece and leaving 8 meters free. If you were to cut at the 4-meter and 6-meter spots, you would indeed have the 2-meter piece as needed, but this cutting would leave you two 4-meter leftover pieces. The largest available piece now is 4 meters, but the total amount free would be 8 meters. This condition is classified as external fragmentation.

**Checkpoint 8.3:** The largest contiguous part of the disk is 8 blocks. So the largest new file can have  $8 \times 512$  bytes of data (4096 bytes). This is less than the available 16 free blocks, therefore there is external fragmentation.

**Checkpoint 8.4:** First fit would put the file in block 1 (block 0 has the directory). Best fit would put the file in block 10, because it is the smallest free space that is big enough. Worst fit would put it in block 14, because it is the largest free space.

**Checkpoint 8.5:** A gibibyte is  $2^{30}$  bytes. Each sector is  $2^{12}$  bytes, so there are  $2^{18}$  sectors. So you need  $2^{18}$  bits in the table, one for each sector. There are  $2^3$  bits in a byte, so the table should be  $2^{15}$  (32768) bytes long.

**Checkpoint 8.6:** 2 Gibibytes is  $2^{31}$  bytes. 512 bytes is  $2^9$  bytes.  $31-9 = 22$ , so it would take 22 bits to store the block number.

**Checkpoint 8.7:** 2 Gibibytes is  $2^{31}$  bytes. 32k bytes is  $2^{15}$  bytes.  $31-15 = 16$ , so it would take 16 bits to store the block number.

**Checkpoint 8.8:** There are 16 free blocks, they can all be linked together to create one new file. This means there is no external fragmentation.

**Checkpoint 8.9:** There are many answers. One answer is you could store a byte count in the directory. Another answer is you could store a byte count in each block.

**Checkpoint 8.10:**  $16+9=25$ .  $2^{25}$  is 32 Mebibytes, which is the largest possible disk.

**Checkpoint 8.11:** There are  $2^{31}/2^{10}=2^{21}$  blocks, so the 21-bit block address will be stored as a 32-bit number. One can store  $1024/4=256$  index entries in one 1024-byte block. So the maximum file size is  $256*1024 = 2^8*2^{10} = 2^{18} = 256$  kibibytes. You can increase the block size or store the index in multiple blocks.

**Checkpoint 8.12:** There are 15 free blocks, and they can create an index table using all the free blocks to create one new file. This means there is no external fragmentation.

**Checkpoint 8.13:** There are 15 free blocks, they can create FAT using all the free blocks to create one new file. Each block is 512 bytes, so the largest file is 15 time 512 bytes; there is no external fragmentation.

**Checkpoint 8.14:** Each directory entry now requires 10 bytes. You could have 50 files, leaving some space for the free space management.

**Checkpoint 8.15:** Change the 1024 to 4096.

**Checkpoint 9.1:** Most people communicate in half-duplex. Normally, when we are talking, the sound of our voice overwhelms our ears, so we usually cannot listen while we are talking.

**Checkpoint 9.2:** Since information is encoded as energy, and data is transferred at a fixed rate, each energy packet will exist for a finite time. Energy per time is power.

**Checkpoint 9.3:** If the units of a signal  $x$  is something like volts or watts, we cannot take the  $\log_{10}(x)$ , because the units of  $\log_{10}(x)$  is not defined. Whenever we use the  $\log_{10}$  to calculate the amplitude of a signal, we always perform the logarithm on a value without dimensions. In other words, we always perform the logarithm on a ratio of one signal to another.

**Checkpoint 9.4:** The performance measure for a storage system is information density in bits/cm<sup>3</sup>.

**Checkpoint 9.5:** With open collector outputs, the low will dominate over HiZ. The signal will be low.

**Checkpoint 10.1:** The  $V_{OL}$  of the 7406 at 40 mA will be 0.7V. This means there will be 4.3V across the coil.

**Checkpoint 10.2:** If they are too close, then the system can turn on-off-on-off-... very quickly, causing the electromagnetic relays to prematurely fail. If they are too far apart, then the system will oscillate with large positive and negative errors.

**Checkpoint 10.3:** Every interrupt, the actuator would be increased or decreased, causing a lot of output changes.

**Checkpoint 10.4:** If the interrupt period were too small, the actuator would be increased to maximum or decreased to minimum, causing it to behave like a bang-bang controller. Basically, the plant would not have time to react to changes in the actuator.

**Checkpoint 10.5:** The output will saturate. The error increases to a very large positive value or decreases down to a very large negative value.

**Checkpoint 10.6:** The limit of the discrete integral as  $\Delta t$  goes to zero is the continuous integral.

**Checkpoint 10.7:** The limit of the discrete derivative as  $\Delta t$  goes to zero is the continuous derivative.

**Checkpoint 10.8:** Yes. Let **watts** be the units of the actuator output and **RPM** be the units of the sensor input. The units of the lag **L** is **sec**. The units of the rate **R** is **cm/sec**. The units of  $\Delta U$  is **watts**.

Proportional  $K_p = 1.2 \Delta U / (L * R)$     **watts/(sec\*(RPM sec)) = watts/ RPM**

Integral             $K_i = 0.5 K_p / L$                     **watts/(RPM-sec)**

Derivative     $K_d = 0.5 K_p L$             **(watts-sec)/RPM**

**Checkpoint 10.9:**  $E = X^* - X$ , so the error is very negative, causing the P term to be very negative, making  $U=100$ . This removes power and gravity will force it down.

**Checkpoint 10.10:**  $SlowDown = WayTooFast + SpeedingUp * LittleBitFast = 50 + (40 * 60) = 50$

The true engineering experience occurs not with your eyes and ears, but rather with your fingers and elbows. In other words, engineering education does not happen by listening in class or reading a book; rather it happens by designing under the watchful eyes of a patient mentor. So, go build something today, then show it to someone you respect!

# Reference Material

Vector address	Number	IRQ	ISR name in <b>Startup.s</b>	NVIC	Priority bits
0x00000038	14	-2	<b>PendSV_Handler</b>	NVIC_SYS_PRI3_R	23 – 21
0x0000003C	15	-1	<b>SysTick_Handler</b>	NVIC_SYS_PRI3_R	31 – 29
0x00000040	16	0	<b>GPIOPortA_Handler</b>	NVIC_PRI0_R	7 – 5
0x00000044	17	1	<b>GPIOPortB_Handler</b>	NVIC_PRI0_R	15 – 13
0x00000048	18	2	<b>GPIOPortC_Handler</b>	NVIC_PRI0_R	23 – 21
0x0000004C	19	3	<b>GPIOPortD_Handler</b>	NVIC_PRI0_R	31 – 29
0x00000050	20	4	<b>GPIOPortE_Handler</b>	NVIC_PRI1_R	7 – 5
0x00000054	21	5	<b>UART0_Handler</b>	NVIC_PRI1_R	15 – 13
0x00000058	22	6	<b>UART1_Handler</b>	NVIC_PRI1_R	23 – 21
0x0000005C	23	7	<b>SSI0_Handler</b>	NVIC_PRI1_R	31 – 29
0x00000060	24	8	<b>I2C0_Handler</b>	NVIC_PRI2_R	7 – 5
0x00000064	25	9	<b>PWMFault_Handler</b>	NVIC_PRI2_R	15 – 13
0x00000068	26	10	<b>PWM0_Handler</b>	NVIC_PRI2_R	23 – 21
0x0000006C	27	11	<b>PWM1_Handler</b>	NVIC_PRI2_R	31 – 29
0x00000070	28	12	<b>PWM2_Handler</b>	NVIC_PRI3_R	7 – 5
0x00000074	29	13	<b>Quadrature0_Handler</b>	NVIC_PRI3_R	15 – 13
0x00000078	30	14	<b>ADC0_Handler</b>	NVIC_PRI3_R	23 – 21
0x0000007C	31	15	<b>ADC1_Handler</b>	NVIC_PRI3_R	31 – 29
0x00000080	32	16	<b>ADC2_Handler</b>	NVIC_PRI4_R	7 – 5
0x00000084	33	17	<b>ADC3_Handler</b>	NVIC_PRI4_R	15 – 13
0x00000088	34	18	<b>WDT_Handler</b>	NVIC_PRI4_R	23 – 21
0x0000008C	35	19	<b>Timer0A_Handler</b>	NVIC_PRI4_R	31 – 29
0x00000090	36	20	<b>Timer0B_Handler</b>	NVIC_PRI5_R	7 – 5
0x00000094	37	21	<b>Timer1A_Handler</b>	NVIC_PRI5_R	15 – 13
0x00000098	38	22	<b>Timer1B_Handler</b>	NVIC_PRI5_R	23 – 21
0x0000009C	39	23	<b>Timer2A_Handler</b>	NVIC_PRI5_R	31 – 29
0x000000A0	40	24	<b>Timer2B_Handler</b>	NVIC_PRI6_R	7 – 5
0x000000A4	41	25	<b>Comp0_Handler</b>	NVIC_PRI6_R	15 – 13
0x000000A8	42	26	<b>Comp1_Handler</b>	NVIC_PRI6_R	23 – 21
0x000000AC	43	27	<b>Comp2_Handler</b>	NVIC_PRI6_R	31 – 29
0x000000B0	44	28	<b>SysCtl_Handler</b>	NVIC_PRI7_R	7 – 5
0x000000B4	45	29	<b>FlashCtl_Handler</b>	NVIC_PRI7_R	15 – 13
0x000000B8	46	30	<b>GPIOPortF_Handler</b>	NVIC_PRI7_R	23 – 21
0x000000BC	47	31	<b>GPIOPortG_Handler</b>	NVIC_PRI7_R	31 – 29
0x000000C0	48	32	<b>GPIOPortH_Handler</b>	NVIC_PRI8_R	7 – 5
0x000000C4	49	33	<b>UART2_Handler</b>	NVIC_PRI8_R	15 – 13
0x000000C8	50	34	<b>SSI1_Handler</b>	NVIC_PRI8_R	23 – 21
0x000000CC	51	35	<b>Timer3A_Handler</b>	NVIC_PRI8_R	31 – 29
0x000000D0	52	36	<b>Timer3B_Handler</b>	NVIC_PRI9_R	7 – 5
0x000000D4	53	37	<b>I2C1_Handler</b>	NVIC_PRI9_R	15 – 13
0x000000D8	54	38	<b>Quadrature 1_Handler</b>	NVIC_PRI9_R	23 – 21
0x000000DC	55	39	<b>CAN0_Handler</b>	NVIC_PRI9_R	31 – 29
0x000000E0	56	40	<b>CAN1_Handler</b>	NVIC_PRI10_R	7 – 5
0x000000E4	57	41	<b>CAN2_Handler</b>	NVIC_PRI10_R	15 – 13
0x000000E8	58	42	<b>Ethernet_Handler</b>	NVIC_PRI10_R	23 – 21
0x000000EC	59	43	<b>Hibernate_Handler</b>	NVIC_PRI10_R	31 – 29
0x000000F0	60	44	<b>USB0_Handler</b>	NVIC_PRI11_R	7 – 5



0x000000F4	61	45	PWM3_Handler	NVIC_PRI11_R	15 – 13
0x000000F8	62	46	uDMA_Handler	NVIC_PRI11_R	23 – 21
0x000000FC	63	47	uDMA_Error	NVIC_PRI11_R	31 – 29

Table 2.6. Some of the interrupt vectors for the TM4C.

## Memory access instructions

**LDR Rd, [Rn]** ; load 32-bit number at [Rn] to Rd  
**LDR Rd, [Rn,#off]** ; load 32-bit number at [Rn+off] to Rd  
**LDR Rd, [Rn,#off]!** ; load 32-bit number at [Rn+off] to Rd, preindex  
**LDR Rd, [Rn],#off** ; load 32-bit number at [Rn] to Rd, postindex  
**LDRT Rd, [Rn,#off]** ; load 32-bit number unprivileged  
**LDR Rd, =value** ; set Rd equal to any 32-bit value (PC rel)  
**LDRH Rd, [Rn]** ; load unsigned 16-bit at [Rn] to Rd  
**LDRH Rd, [Rn,#off]** ; load unsigned 16-bit at [Rn+off] to Rd  
**LDRH Rd, [Rn,#off]!** ; load unsigned 16-bit at [Rn+off] to Rd, pre  
**LDRH Rd, [Rn],#off** ; load unsigned 16-bit at [Rn] to Rd, postindex  
**LDRHT Rd, [Rn,#off]** ; load unsigned 16-bit unprivileged  
**LDRSH Rd, [Rn]** ; load signed 16-bit at [Rn] to Rd  
**LDRSH Rd, [Rn,#off]** ; load signed 16-bit at [Rn+off] to Rd  
**LDRSH Rd, [Rn,#off]!** ; load signed 16-bit at [Rn+off] to Rd, pre  
**LDRSH Rd, [Rn],#off** ; load signed 16-bit at [Rn] to Rd, postindex  
**LDRSHT Rd, [Rn,#off]** ; load signed 16-bit unprivileged  
**LDRB Rd, [Rn]** ; load unsigned 8-bit at [Rn] to Rd  
**LDRB Rd, [Rn,#off]** ; load unsigned 8-bit at [Rn+off] to Rd  
**LDRB Rd, [Rn,#off]!** ; load unsigned 8-bit at [Rn+off] to Rd, pre  
**LDRB Rd, [Rn],#off** ; load unsigned 8-bit at [Rn] to Rd, postindex  
**LDRBT Rd, [Rn,#off]** ; load unsigned 8-bit unprivileged  
**LDRSB Rd, [Rn]** ; load signed 8-bit at [Rn] to Rd  
**LDRSB Rd, [Rn,#off]** ; load signed 8-bit at [Rn+off] to Rd  
**LDRSB Rd, [Rn,#off]!** ; load signed 8-bit at [Rn+off] to Rd, pre  
**LDRSB Rd, [Rn],#off** ; load signed 8-bit at [Rn] to Rd, postindex  
**LDRSBT Rd, [Rn,#off]** ; load signed 8-bit unprivileged  
**LDRD Rd,Rd2,[Rn,#off]** ; load 64-bit at [Rn+off] to Rd,Rd2  
**LDRD Rd,Rd2,[Rn,#off]!** ; load 64-bit at [Rn+off] to Rd,Rd2,pre  
**LDRD Rd,Rd2,[Rn],#off** ; load 64-bit at [Rn] to Rd,Rd2, postindex  
**LDMFD Rn{!}, Reglist** ; load reg from list at Rn(inc), !update Rn  
**LDMIA Rn{!}, Reglist** ; load reg from list at Rn(inc), !update Rn  
**LDMDB Rn{!}, Reglist** ; load reg from list at Rn(dec), !update Rn  
**STMIA Rn{!}, Reglist** ; store reg from list to Rn(inc), !update Rn  
**STMFDB Rn{!}, Reglist** ; store reg from list to Rn(dec), !update Rn  
**STMDB Rn{!}, Reglist** ; store reg from list to Rn(dec), !update Rn  
**STR Rt, [Rn]** ; store 32-bit Rt to [Rn]  
**STR Rt, [Rn,#off]** ; store 32-bit Rt to [Rn+off]  
**STR Rt, [Rn,#off]!** ; store 32-bit Rt to [Rn+off], pre  
**STR Rt, [Rn],#off** ; store 32-bit Rt to [Rn], postindex

**STRT Rt, [Rn,#off]** ; store 32-bit Rt to [Rn+off] unprivileged  
**STRH Rt, [Rn]** ; store least sig. 16-bit Rt to [Rn]  
**STRH Rt, [Rn,#off]** ; store least sig. 16-bit Rt to [Rn+off]  
**STRH Rt, [Rn,#off]!** ; store least sig. 16-bit Rt to [Rn+off], pre  
**STRH Rt, [Rn],#off** ; store least sig. 16-bit Rt to [Rn], postindex  
**STRHT Rt, [Rn,#off]** ; store least sig. 16-bit unprivileged  
**STRB Rt, [Rn]** ; store least sig. 8-bit Rt to [Rn]  
**STRB Rt, [Rn,#off]** ; store least sig. 8-bit Rt to [Rn+off]  
**STRB Rt, [Rn,#off]!** ; store least sig. 8-bit Rt to [Rn+off],pre  
**STRB Rt, [Rn],#off** ; store least sig. 8-bit Rt to [Rn], postindex  
**STRBT Rt, [Rn,#off]** ; store least sig. unprivileged  
**STRD Rd,Rd2,[Rn,#off]** ; store 64-bit Rd,Rd2 to [Rn+off]  
**STRD Rd,Rd2,[Rn,#off]!** ; store 64-bit Rd,Rd2 to [Rn+off], pre  
**STRD Rd,Rd2,[Rn],#off** ; store 64-bit Rd,Rd2 to [Rn], postindex  
**PUSH Reglist** ; push 32-bit registers onto stack  
**POP Reglist** ; pop 32-bit numbers from stack into registers  
**ADR Rd, label** ; set Rd equal to the address at label  
**MOV{S} Rd, <op2>** ; set Rd equal to op2  
**MOV Rd, #im16** ; set Rd equal to im16, im16 is 0 to 65535  
**MOVT Rd, #im16** ; set Rd bits 31-16 equal to im16  
**MVN{S} Rd, <op2>** ; set Rd equal to -op2

## Branch instructions

**B label** ; branch to label Always  
**BEQ label** ; branch if Z == 1 Equal  
**BNE label** ; branch if Z == 0 Not equal  
**BCS label** ; branch if C == 1 Higher or same, unsigned  $\geq$   
**BHS label** ; branch if C == 1 Higher or same, unsigned  $\geq$   
**BCC label** ; branch if C == 0 Lower, unsigned  $<$   
**BLO label** ; branch if C == 0 Lower, unsigned  $<$   
**BMI label** ; branch if N == 1 Negative  
**BPL label** ; branch if N == 0 Positive or zero  
**BVS label** ; branch if V == 1 Overflow  
**BVC label** ; branch if V == 0 No overflow  
**BHI label** ; branch if C==1 and Z==0 Higher, unsigned  $>$   
**BLS label** ; branch if C==0 or Z==1 Lower or same, unsigned  $\leq$   
**BGE label** ; branch if N == V Greater than or equal, signed  $\geq$   
**BLT label** ; branch if N != V Less than, signed  $<$   
**BGT label** ; branch if Z==0 and N==V Greater than, signed  $>$   
**BLE label** ; branch if Z==1 or N!=V Less than or equal, signed  $\leq$   
**BX Rm** ; branch indirect to location specified by Rm  
**BL label** ; branch to subroutine at label  
**BLX Rm** ; branch to subroutine indirect specified by Rm  
**CBNZ Rn,label** ; branch if Rn not zero

CBZ Rn,label ; branch if Rn zero  
IT{x{y}{z}}cond ; if then block with x,y,z T(true) or F(false)  
TBB [Rn, Rm] ; table branch byte  
TBH [Rn, Rm, LSL #1] ; table branch halfword

### Mutual exclusive instructions

CLREX ; clear exclusive  
LDREX{cond} Rt,[Rn{,#offset}] ; load 32-bit exclusive  
STREX{cond} Rd,Rt,[Rn{,#offset}] ; store 32-bit exclusive  
LDREXB{cond} Rt,[Rn] ; load 8-bit exclusive  
STREXB{cond} Rd,Rt,[Rn] ; store 8-bit exclusive  
LDREXH{cond} Rt,[Rn] ; load 16-bit exclusive  
STREXH{cond} Rd,Rt,[Rn] ; store 16-bit exclusive

### Miscellaneous instructions

BKPT #imm ; execute breakpoint, debug state 0 to 255  
CPSIE F ; clear faultmask F=0  
CPSIE I ; enable interrupts (I=0)  
CPSID F ; set faultmask F=1  
CPSID I ; disable interrupts (I=1)  
DMB ; data memory barrier, memory access to finish  
DSB ; data synchronization barrier, instructions to finish  
ISB ; instruction synchronization barrier, finish pipeline  
MRS Rd,SpecReg ; move special register to Rd  
MSR Rd,SpecReg ; move Rd to special register  
NOP ; no operation  
SEV ; Send Event  
SVC #im8 ; supervisor call (0 to 255)  
WFE ; wait for event  
WFI ; wait for interrupt

### Logical instructions

AND{S} {Rd,} Rn, <op2> ; Rd=Rn&op2 (op2 is 32 bits)  
BFC Rd,#lsb,#width ; clear bits in Rn  
BFI Rd,Rn,#lsb,#width ; bit field insert, Rn into Rd  
ORR{S} {Rd,} Rn, <op2> ; Rd=Rn|op2 (op2 is 32 bits)  
EOR{S} {Rd,} Rn, <op2> ; Rd=Rn^op2 (op2 is 32 bits)  
BIC{S} {Rd,} Rn, <op2> ; Rd=Rn&(~op2) (op2 is 32 bits)  
ORN{S} {Rd,} Rn, <op2> ; Rd=Rn|(~op2) (op2 is 32 bits)  
TST Rn, <op2> ; Rn&op2 (op2 is 32 bits)  
TEQ Rn, <op2> ; Rn^op2 (op2 is 32 bits)  
LSR{S} Rd, Rm, Rs ; logical shift right Rd=Rm>>Rs (unsigned)  
LSR{S} Rd, Rm, #n ; logical shift right Rd=Rm>>n (unsigned)

**ASR{S} Rd, Rm, Rs** ; arithmetic shift right  $Rd=Rm \gg Rs$  (signed)  
**ASR{S} Rd, Rm, #n** ; arithmetic shift right  $Rd=Rm \gg n$  (signed)  
**LSL{S} Rd, Rm, Rs** ; shift left  $Rd=Rm \ll Rs$  (signed, unsigned)  
**LSL{S} Rd, Rm, #n** ; shift left  $Rd=Rm \ll n$  (signed, unsigned)  
**REV Rd, Rn** ; Reverse byte order in a word  
**REV16 Rd, Rn** ; Reverse byte order in each halfword  
**REVSH Rd, Rn** ; Reverse byte order in the bottom halfword,  
; and sign extends to 32 bits  
**RBIT Rd, Rn** ; Reverse the bit order in a 32-bit word  
**SBFX Rd,Rn,#lsb,#width** ; signed bit field and extract  
**UBFX Rd,Rn,#lsb,#width** ; unsigned bit field and extract  
**SXTB {Rd,}Rm{,ROR #n}** ; Sign extend byte  
**SXTH {Rd,}Rm{,ROR #n}** ; Sign extend halfword  
**UXTB {Rd,}Rm{,ROR #n}** ; Zero extend byte  
**UXTH {Rd,}Rm{,ROR #n}** ; Zero extend halfword

#### Arithmetic instructions

**ADD{S} {Rd,} Rn, <op2>** ;  $Rd = Rn + op2$   
**ADD{S} {Rd,} Rn, #im12** ;  $Rd = Rn + im12$ , im12 is 0 to 4095  
**CLZ Rd, Rm** ;  $Rd =$  number of leading zeros in Rm  
**SUB{S} {Rd,} Rn, <op2>** ;  $Rd = Rn - op2$   
**SUB{S} {Rd,} Rn, #im12** ;  $Rd = Rn - im12$ , im12 is 0 to 4095  
**RSB{S} {Rd,} Rn, <op2>** ;  $Rd = op2 - Rn$   
**RSB{S} {Rd,} Rn, #im12** ;  $Rd = im12 - Rn$   
**CMP Rn, <op2>** ;  $Rn - op2$  sets the NZVC bits  
**CMN Rn, <op2>** ;  $Rn - (-op2)$  sets the NZVC bits  
**MUL{S} {Rd,} Rn, Rm** ;  $Rd = Rn * Rm$  signed or unsigned  
**MLA Rd, Rn, Rm, Ra** ;  $Rd = Ra + Rn * Rm$  signed or unsigned  
**MLS Rd, Rn, Rm, Ra** ;  $Rd = Ra - Rn * Rm$  signed or unsigned  
**UDIV {Rd,} Rn, Rm** ;  $Rd = Rn / Rm$  unsigned  
**SDIV {Rd,} Rn, Rm** ;  $Rd = Rn / Rm$  signed  
**UMULL RdLo,RdHi,Rn,Rm** ; Unsigned long multiply 32by32 into 64  
**UMLAL RdLo,RdHi,Rn,Rm** ; Unsigned long multiply, with accumulate  
**SMULL RdLo,RdHi,Rn,Rm** ; Signed long multiply 32by32 into 64  
**SMLAL RdLo,RdHi,Rn,Rm** ; Signed long multiply, with accumulate  
**SSAT Rd,#n,Rm{,shift #s}** ; signed saturation to n bits  
**USAT Rd,#n,Rm{,shift #s}** ; unsigned saturation to n bits

**Notes** Ra Rd Rm Rn Rt represent 32-bit registers

value any 32-bit value: signed, unsigned, or address

{S} if S is present, instruction will set condition codes

**#im8** any value from 0 to 255  
**#im12** any value from 0 to 4095  
**#im16** any value from 0 to 65535  
**{Rd,}** if Rd is present Rd is destination, otherwise Rn  
**#n** any value from 0 to 31  
**#off** any value from -255 to 4095  
**label** any address within the ROM of the microcontroller  
**SpecReg** APSR,IPSR,EPSR,IEPSR,IAPSR,EAPSR,PSR,MSP,PSP,  
 PRIMASK,BASEPRI,BASEPRI\_MAX,FAULTMASK, or CONTROL.  
**Reglist** is a list of registers. E.g., {R1,R3,R12}  
**op2** the value generated by <op2>

Examples of flexible operand <op2> creating the 32-bit number. E.g., **Rd = Rn+op2**

**ADD Rd, Rn, Rm ; op2 = Rm**  
**ADD Rd, Rn, Rm, LSL #n ; op2 = Rm<<n Rm is signed, unsigned**  
**ADD Rd, Rn, Rm, LSR #n ; op2 = Rm>>n Rm is unsigned**  
**ADD Rd, Rn, Rm, ASR #n ; op2 = Rm>>n Rm is signed**  
**ADD Rd, Rn, #constant ; op2 = constant**, where X and Y are hexadecimal digits:
 

- produced by shifting an 8-bit unsigned value left by any number of bits
- in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**

Parameter	PN2222 ( $I_C=150\text{mA}$ ) PN2907 ( $I_C=150\text{mA}$ )	2N2222 ( $I_C=500\text{mA}$ ) 2N2907 ( $I_C=500\text{mA}$ )	TIP120 ( $I_C=3\text{A}$ ) TIP125 ( $I_C=3\text{A}$ )
$h_{fe}$	100	40	1000
$V_{BEsat}$	0.6	2	2.5 V
$V_{CE}$ at saturation	0.3	1	2 V

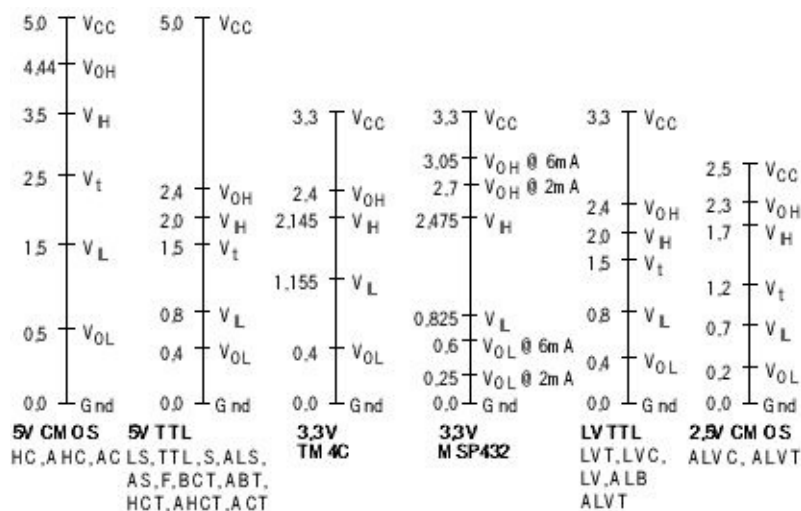
*Design parameters for the 2N2222 and TIP120.*

Chip	Current	Comment
L293D	0.6 A	Dual, diodes
L293	1 A	Dual
DRV8848	2 A	Dual, fault status
TPIC0107	3 A	Direction, fault status

*H-bridge drivers*

Family	Example	$I_{OH}$	$I_{OL}$	$I_{IH}$	$I_{IL}$
Standard TTL	7404	0.4 mA	16 mA	40 $\mu$ A	1.6 mA
Low Power Schottky	74LS04	0.4 mA	4 mA	20 $\mu$ A	0.4 mA
High Speed CMOS	74HC04	4 mA	4 mA	1 $\mu$ A	1 $\mu$ A
Adv High Speed CMOS	74AHC04	4 mA	4 mA	1 $\mu$ A	1 $\mu$ A
MSP432 regular drive	MSP432	6 mA	6 mA	20 nA	20 nA
MSP432 high drive	MSP432	20 mA	20 mA	20 nA	20 nA
TM4C 2mA-drive	TM4C123	2 mA	2 mA	2 $\mu$ A	2 $\mu$ A
TM4C 4mA-drive	TM4C123	4 mA	4 mA	2 $\mu$ A	2 $\mu$ A
TM4C 8mA-drive	TM4C123	8 mA	8 mA	2 $\mu$ A	2 $\mu$ A
TM4C 12mA-drive	TM4C1294	12 mA	12 mA	2 $\mu$ A	2 $\mu$ A

*The input and output currents of various digital logic families and microcontrollers.*



*Voltage thresholds for various digital logic families.*